# Optimising Parallel Logic Programming Systems for Scalable Machines

Vítor Santos Costa and Ricardo Bianchini

Technical Report Series: DCC-97-7

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

---

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

# Optimising Parallel Logic Programming Systems for Scalable Machines

Vítor Santos Costa

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

Ricardo Bianchini

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

October 1, 1997

**Abstract**

Logic programs are good examples of symbolic applications that often exhibit large amounts of implicit parallelism and that can greatly benefit from parallel computers. Parallel logic programming (PLP) systems have obtained excellent results for traditional bus-based shared-memory architectures. However, the scalable multiprocessors being developed today pose new challenges, such as the high latency of memory accesses and the demand for scalability. Our experience with a sophisticated PLP system, Andorra-I, demonstrates that indeed performance suffers greatly on modern architectures. This paper addresses the question of whether the poor performance 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 perform a detailed analysis of the cache behaviour of all Andorra-I data structures via execution-driven simulation of a DASH-like multiprocessor. Based on this analysis we optimise the Andorra-I code using 5 different techniques. We present the isolated and combined performance improvements provided by these optimisations. The overall results we obtained are quite impressive. A few of our logic programs begun to approach linear speedup as a result of our optimisations. In fact, for one of our programs 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 PLP systems are indeed complex and sometimes irregular, these systems can and should perform well on modern scalable multiprocessors.

**Keywords:** Scalable Multiprocessors, Parallel Logic Programming, And-Or Parallelism, Performance Evaluation.

## 1 Introduction

Parallel computers can improve performance of both numerical and symbolic applications. Logic programs are good examples of symbolic applications that often exhibit large amounts of implicit parallelism and that can greatly benefit from parallel computers. To exploit this parallelism fully, parallel logic programming (PLP) 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.

Several PLP systems have been developed so far and have obtained good results for traditional bus-based shared-memory architectures, such as the Sequent Symmetry. However, the scalable multiprocessors being developed today pose new challenges, such as the high latency of memory

accesses and the demand for scalability. Hence, the performance of these multiprocessors depends heavily on the processors' cache miss rates and may be limited by the communication overhead involved in data sharing.

The complexity of PLP systems and the large amount of data they process raise the issue of whether PLP systems can obtain good performance on these new parallel architectures. In order to address this issue, we experiment with a sophisticated PLP system, Andorra-I [18], that exploits both dependent and-parallelism and or-parallelism. Andorra-I is a particularly interesting example of PLP system, since most of its data structures are similar to the ones of several other PLP systems.

Andorra-I has obtained good performance on traditional bus-based machines [26], but our experience with it running on modern multiprocessors [17] demonstrates that scalability indeed suffers greatly on these architectures.

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 cache 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 of the misses.

Based on this analysis we remove the main performance limiting factors in Andorra-I through a small set of optimisations that did not require a redesign of the system. More specifically, we optimise Andorra-I using 5 different techniques: elimination of unnecessary shared variables, data layout modification, privatisation of shared data structures, lock distribution, and elimination of unnecessary 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, the elimination of unnecessary shared variables and the modification of the data layout produced the greatest improvements. The improvements achieved when all optimisation techniques are combined are quite impressive. A few of our logic programs begun to 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 PLP systems are indeed complex and sometimes irregular, these systems can and should scale well on modern scalable multiprocessors.

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, how Andorra-I was ported to the simulator, and the description and 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 compares to related work and section 7 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 [24]. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [18, 26]. 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 alternative execution paths are eliminated. The Andorra-I system also exploits or-parallelism that arises from the non-determinate goals. Its implementation is influenced by JAM [6] when exploiting and-parallelism, and by Aurora [14] when exploiting or-parallelism.

The Andorra-I system consists of several components. In this work we are mostly interested in the *engine*, that is responsible for the execution of the Andorra-I programs [27], and in the *schedulers*, that manage and- and or-work.

A processing element that performs computation in Andorra-I is called a *worker*. In practice, each worker corresponds to a separate processor. Workers are organised into *teams*, such that different teams of workers cooperate to share or-work. Workers in a team cooperate with each other in order to share available and-work. Each team has a single master and zero or more slaves. The master is responsible for general team management and synchronisation. Also, if no determinate goals are available to the team, the team must select a non-determinate goal. In Andorra-I it is up to the master to select the goal, take an alternative from the goal, and create a *choicepoint* to represent the availability of other alternatives. Note that workers within a team share the same set of variables, as they all are in the same branch of the search tree.

Most of the execution time of workers should be spent executing *engine* code [27], i.e. performing reductions. 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 implied by the logic program. Our experiments used the Bristol or-scheduler [2], originally developed for the Aurora system. 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 [8] dynamically arranges workers into teams in a way that allows both and- and or-parallelism to be freely exploited when they are available.

Andorra-I operation requires access to shared memory for both execution and scheduling. 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 for each choicepoint, 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.
- **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 [27]). 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 [23] 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.

## 2.2 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [13], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, a 128-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 [22], developed by Veenstra and Fowler, to simulate the MIPS architecture, and a back-end, developed by Bianchini and Veenstra, to simulate the memory and interconnection systems.

We use a write-invalidate (WI) protocol in order to keep caches coherent. In this protocol, whenever a processor writes a data item, copies of the cache block containing the item in other processors' caches are invalidated. If one of the invalidated processors later requires the same item, it will have to fetch it from the writer's cache. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [12]. We classify cache misses under this protocol using the algorithm described in [7], as extended in [4].

## 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 the `binutils-2.6` assembler and linker under a Solaris 2.4 environment as cross development tools for this purpose. Andorra-I was compiled with `-O2`.

Most of the port was straightforward. The only difficulties arose with shared memory allocation and locking. For shared memory allocation we use the `shmalloc` library supported by MINT. For locking in modern MIPS machines, we would use the `ll`, `sc`, and `sync` machine instructions [10] 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 *fetch_and_phi* instructions.

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

# 3 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 [5], `bt-cluster`, and a program to calculate approximate solutions to the traveling salesman 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 salesman program is based on a Reform Prolog [3] benchmark that finds an approximate solution for the TSP problem in a graph with 24 nodes. To obtain best performance, we rewrote the original applications to become a determinate-only computation.

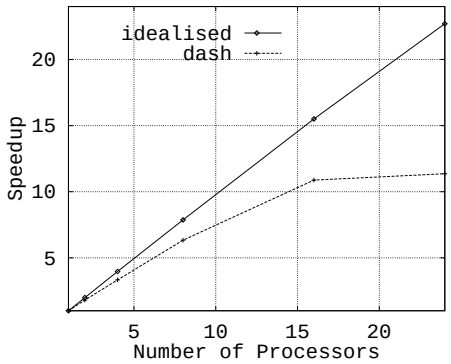


Figure 1: Speedups for `bt-cluster`

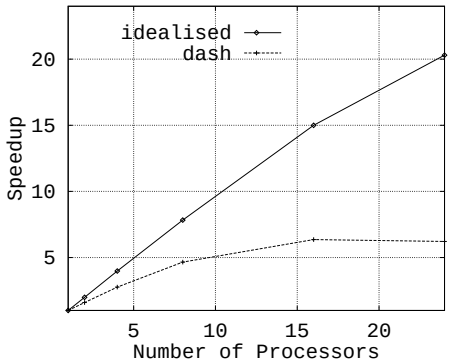


Figure 2: Speedups for `tsp`

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 could 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 and less than 50% for 24 processors.

Figure 2 shows that the `tsp` application has worse performance than `bt-cluster`. The maximum speedup is only 6.3 for the 16-processor configuration, and actually decreases for 24 processors, whereas the ideal machine would achieve speedups of 15 for 16 processors and of 20 for 24 processors.

The simulation results show a significant increase in read and write latencies for the applications, as we increase the number of processors. In the case of `bt-cluster` the average memory access latency more than doubles from 2.8 processor cycles with 2 processors to 5.8 cycles with 16 processors. In the case of `tsp` the average latency increases even more significantly, from 5.2 cycles on 2 processors to 17.9 cycles on 16 processors.

The average latency of memory accesses depends heavily on the cache miss rate and on the congestion caused by the access requests and replies. The miss rate is not terribly high for the two benchmarks, so we conclude that congestion is the main contributor to the poor performance of the programs. For `bt-cluster`, the miss-rate increases from 1.2% for 2 processors up to 1.6% for 16 processors. For `tsp`, the miss-rate increases from 2.8% for 2 processors to 3% for 16 processors. These miss rates are dominated by sharing misses.

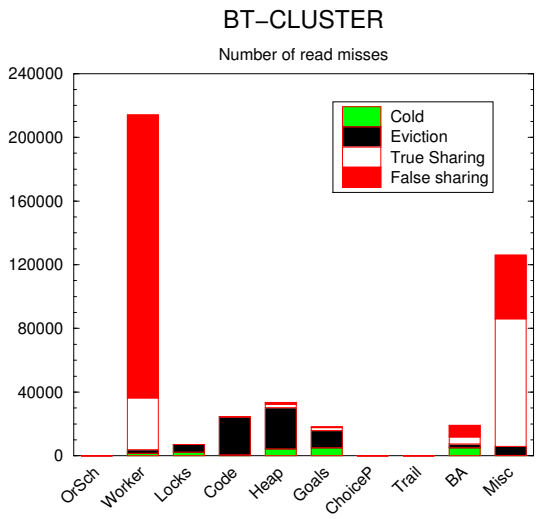


Figure 3: Misses by data area for `bt-cluster`

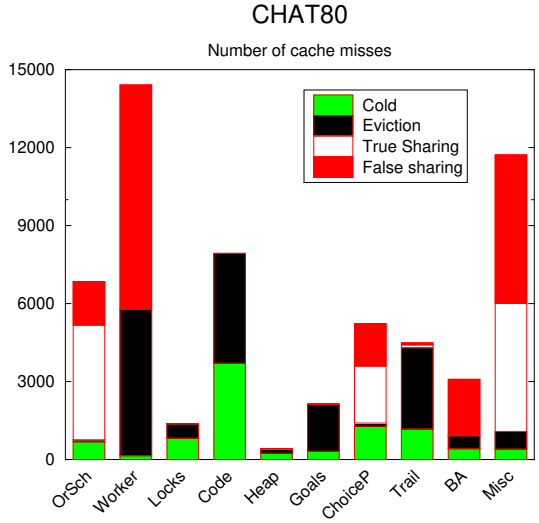


Figure 4: Misses by data area for `chat80`

Figure 3 illustrates the number and sources of cache misses in and-parallel applications. The Figure presents results for `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 is an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren [25]. 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 [11]. The application has much longer running times, and should at least in theory have significant or-parallelism.

Figure 5 shows the speedups for the `chat80` application from 1 to 24 processors. The `idealised` 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 [9]. In contrast, the DASH curve reaches a maximum speedup of 4.2 for 16 processors.

Figure 6 shows the speedups for the `fp` application from 1 to 24 processors. The theoretical speedup is very good, in fact quite close to linear, in sharp contrast to the actual speedup for the DASH-like machine, which barely manages to go up to 6 for 16 processors.

The explanation for the bad speedups under realistic assumptions is again a sharp increase in the average latency of memory accesses. The latency jumps from 4.9 cycles on 2 processors to 15.9 cycles

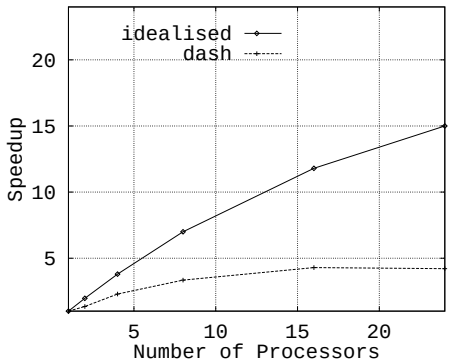


Figure 5: Speedups for `chat80`

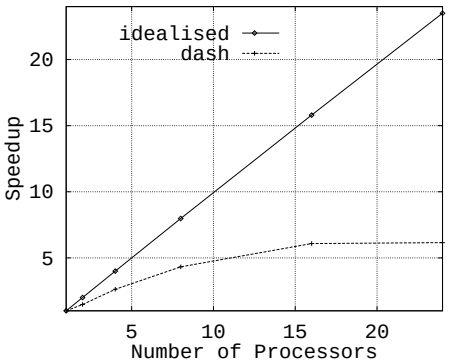


Figure 6: Speedups for `fp`

on 16 processors for `fp`, while increasing from 5.7 cycles on 2 processors to 8.8 cycles on 16 processors for the less memory-intensive `chat80`.

Miss rates exhibit an interesting variation for these two applications. In the case of `chat80` the miss rate actually decreases from 3.4% with 2 processors to 2.2% with 16 processors. This is a common effect resulting from an increase in overall cache space that, in turn, causes a reduction of the number of eviction misses. In the case of `fp` the miss rate increases from 2.4% with 2 processors to a high 4% with 16 processors. In this application, 60% of the misses are caused by sharing, while the other 40% is divided into cold start and eviction misses. False sharing accounts for 50%, while true sharing accounts for only 10% of the misses. Almost all non-sharing misses are evictions.

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 that `chat80` does not have enough parallelism to feed 16 processors. This would suggest that most sharing misses would result from or-parallel scheduling areas, `OrSch` and `ChoicePt`. The figure shows however that the cache misses originate mainly from `Worker` and `Misc` as in the and-parallel applications, indicating again that these two areas should probably be optimised.

Note also the large number of eviction misses in the `Trail`. The trail is used to store conditional bindings, and is used in backtracking and when searching for work. The high number of misses in this area indicates that workers often go here, and suggests work is fine-grained.

**And/Or-Parallel Application.** As an example of and/or 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 [1]. In this technique, the co-routining inherent to 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. To test the program, we ask it to schedule 11 aircrafts, 36 crew members and 10 flights. The degree of and- and or-parallelism in this program varies according to the queries, but this query gives rise to more and-parallelism than or-parallelism. The system uses the reconfigurer to dynamically adapt to the available sources of parallelism.

Figure 7 shows the speedups for `pan2`. The `idealised` curve shows that the application has less parallelism than all other applications; the ideal speedup does not even reach 12 on 24 processors. When run on the `DASH` simulator, `pan2` exhibits unacceptable speedups for all numbers of processors;

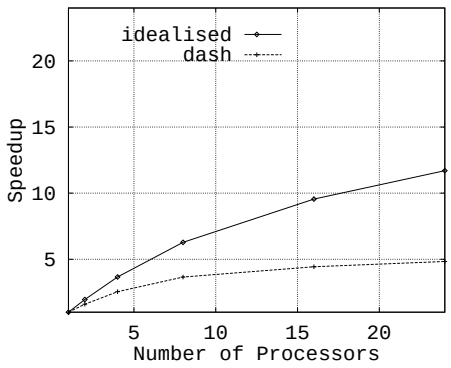


Figure 7: Speedups for `pan2`

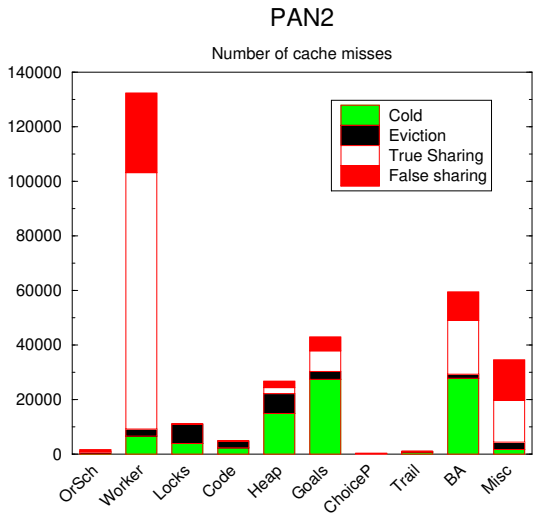


Figure 8: Misses by data area for `pan2`

speedup starts out at about 1.8 for 2 processors and slowly improves to a maximum of only 4.8 for 24 processors.

The main source of performance degradation under realistic assumptions is again a significant increase in the average read and write latency. Average memory access latency goes up from 4 cycles on 2 processors to 11.5 on 16 processors. Figure 8 shows the distribution of cache misses by the different Andorra-I data areas for 16 processors. In this case, the `Worker` area clearly dominates, since the contribution from the `Misc` area is not as significant as in the and-parallel benchmarks. Note that there is more true than false sharing activity in `Worker`. The true sharing probably results from idle processors looking for work.

# 4 Optimisation Techniques and Performance

The previous analysis suggests that the miss rates and network and memory congestion misses may generate are the main reasons for the poor scalability of Andorra-I. It is interesting to note that most misses originate 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, especially 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. The other three were based on our original intuitions regarding the system, and were the ones we would have performed first without simulation data.

## 4.1 Elimination of Unnecessary Variables

In two of our detailed analyses, 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. By investigating each component in the area, we detected that the counters were the major source of

misses. As the counters are only used for research and debugging purposes, we were able to eliminate them from the Andorra-I code.

Figure 9 shows the result of this optimisation for three benchmarks, `bt-cluster`, `chat80` and `pan2`. Other applications behave similarly. The figure plots the variation in speedup as a function of the number of processors.

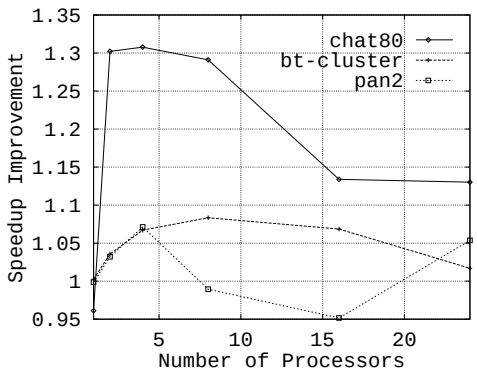


Figure 9: Impact of Counter Elimination Optimisations

The figure shows 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. Updating both the reduction and failure counters seems to badly impact execution. Note that the benefit is greater on small numbers of processors, suggesting that other sources of misses are keeping the memory access latency up.

The optimisation does not impact the and-parallel benchmarks as much. The `bt-cluster` application improves by about 5% for most machine configurations, whereas `pan2` actually experiences speedup degradation for 16 workers. The reason for this effect is that `pan2` includes both and- and or-parallelism and, therefore, is more sensitive to small variations in scheduling.

## 4.2 Data Layout Modification

All benchmarks but `pan2` exhibit a high false sharing rate. On 16 processors, 15% of the misses in `pan2` are false sharing misses, whereas in the other applications false sharing represents between 40% for `chat80` to 51% for `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`, `ChoicePt` and `BA` areas, besides the `Misc` area optimised above.

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 separated, and *field reordering* to separate fields that were physically close but logically distinct. Although these are well-known techniques, padding required careful analysis, as it increases eviction misses significantly. The field reordering technique was not easily applied either, as the relationships between fields are quite complex.

Padding is very dangerous for the dynamic data areas, such as `ChoicePt` and `BA`. This restricted our options for layout modification to just field reordering for these areas.

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 the `BA` 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.

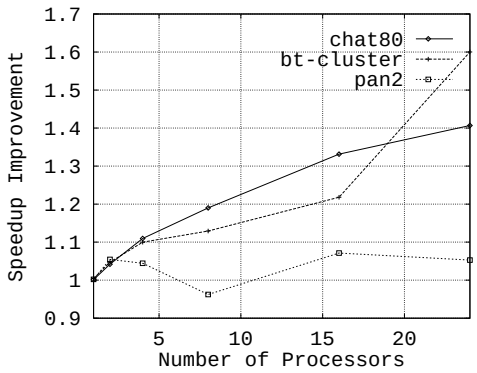


Figure 10: Impact of Data Layout Optimisations

Figure 10 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.

The `pan2` application benefits only slightly from this optimisation. Note that we again have a slowdown, in this case for 4 processors, probably caused by slightly different scheduling.

## 4.3 Privatisation of Shared Variables

It is often the case that shared memory accesses can be avoided by making local copies in each processor. In the best case, the shared variables are read-only and hence local copies can actually be allocated in private memory. This technique is usually called *privatisation*.

The high number of references to `Worker` suggested that privatisation could be applied there. In fact, Andorra-I did already use private copies of the variables in `Worker` and there was little room for improvement.

The `Locks` and `Code` data areas are the major candidates to privatisation in Andorra-I. The `Locks` area only includes pointers to the actual locks, is thus read-only during execution, and can be easily privatised. Another area that is also read-only during parallel execution of our benchmarks is `Code`. Unfortunately, logic programs in general can change the database and, therefore, update `Code`, making privatisation complex. Given that our cache behaviour results indicate that `Code` is only a major source of misses for `chat80`, we did not implement the optimisation for this area.

Figure 11 shows the impact of privatisation the speedup of Andorra-I. The figure shows that privatisation improves speedups by as much as 10% for all benchmarks. Notice that the impact of the optimisation decreases as the number of processors increases.

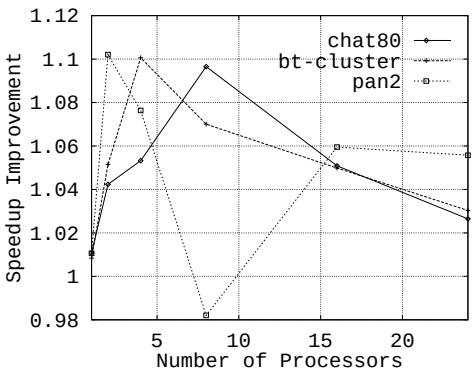


Figure 11: Impact of Privatisation Optimisations

## 4.4 Lock Distribution

Andorra-I requires locking on accesses to logical variables, and-scheduling, or-scheduling, and stack management. The original implementation used a single array of locks to implement these operations. In the worst case, several workers would contend for the same lock causing unnecessary contention.

Our intuition was that this problem could limit system scalability. We therefore tried to improve the system by having different locks data structures for different purposes. This optimisation should be particularly useful for or-parallel applications, as it prevents different teams from contending to access logical variables. The optimisation has a cost, however. If the arrays of locks are not privatised, more shared lock accesses will occur, resulting in additional expensive remote misses.

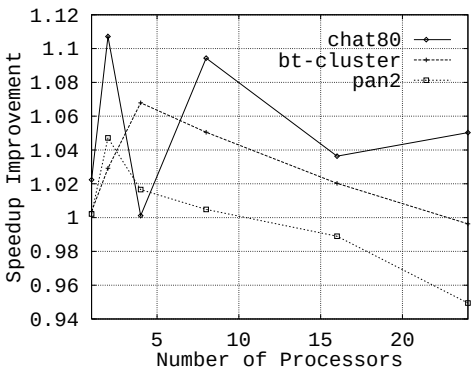


Figure 12: Impact of Lock Distribution Optimisations

Figure 12 shows the impact of lock distribution in the absence of data privatisation. The `bt-cluster` and `chat80` applications benefit somewhat from this optimisation. On the other hand, the `pan2` application already had significant misses from the `Locks` area. Increasing the number of shared locks also increases the number of remote misses and leads to worse speedups on 16 and 24

processors. Note that the number of misses would not have increased, if this optimisation had been applied together with privatisation. Section 5 will present results for the combination of these two optimisations.

### 4.5 Elimination of Locking in Scheduling

The last optimisation we made to Andorra-I was relatively simple: we avoid contention for locks when searching for and-work 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.

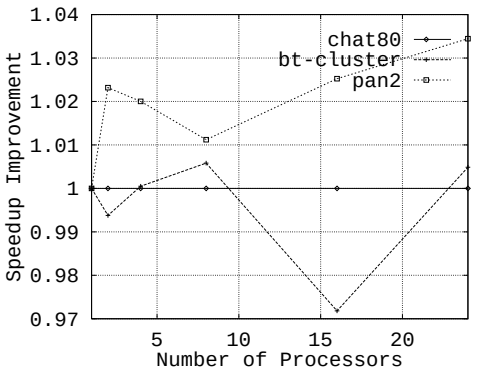


Figure 13: Impact of Scheduling Optimisations

Figure 13 shows the variations in speedup entailed by this optimisation. The figure shows that the locking elimination has no impact on the or-parallel benchmark, `chat80`, as this application never uses the and-scheduler. The optimisation does not improve the speedup of `bt-cluster` either, since this application has plenty of and-work to keep processors busy. Finally, locking elimination provides a small increase in speedup for `pan2`, as it avoids locking when there is no and-work.

### 4.6 Summary of Results

The speedup results presented in the previous subsections show that two optimisation techniques are by far the most profitable for `bt-cluster` and `chat80`: elimination of unnecessary shared variables and data layout modification. No technique was able to significantly improve the speedups of `pan2`.

It is interesting to note that the two most profitable techniques for our applications are exactly the ones that were prompted by our detailed analysis of cache behaviour. The other three techniques were based on our original intuitions resulting from years of experience with Andorra-I.

## 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.

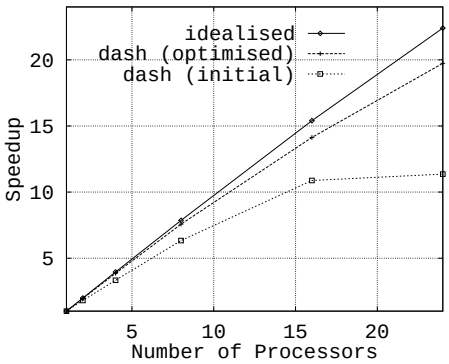


Figure 14: Speedups for `bt-cluster`

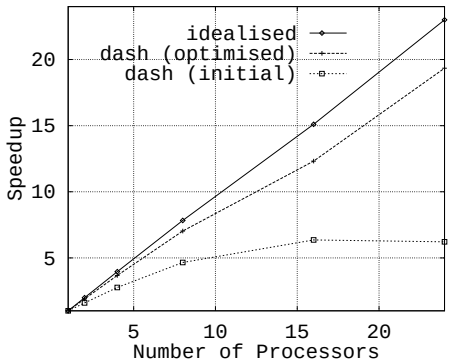


Figure 15: Speedups for `tsp`

**And-Parallel Benchmarks** We first discuss the and-parallel benchmarks.

Figure 14 and 15 show the speedups for the two and-parallel applications running on top of the modified Andorra-I system, as a function of the number of processors. 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 misses 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 16 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.

**Or-Parallel Benchmarks** The or-parallel benchmarks also show remarkable improvements due to the combination of the optimisation techniques we applied. Figure 18 shows the speedups for `chat80` and Figure 19 shows the speedups for `fp`.

The figure shows that the maximum speedup for `chat80` almost doubles from one version of the system to the other; speedup goes up from 4.2 to 7.4 on 24 processors. Note that speedups for the optimised system still flatten out at 16 workers, but at a much better efficiency. The other 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.

The evolution of the cache miss rates is consistent with these speedups. The miss rate for `chat80` decreases from 2.2% at 16 processors in the old version to 1.2% in the new version. The miss rate for `fp` has an even more impressive evolution. On 16 processors, the miss rates decreases from 4.1% with the original version to 0.8% in the new version.

Figure 17 shows the distribution of misses for `chat80` with 16 processors. Again, we keep the same Y-axis as before to facilitate comparisons. 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` area remains however. Sharing misses are now concentrated in the `OrSch` and `ChoicePt` areas, as they should.

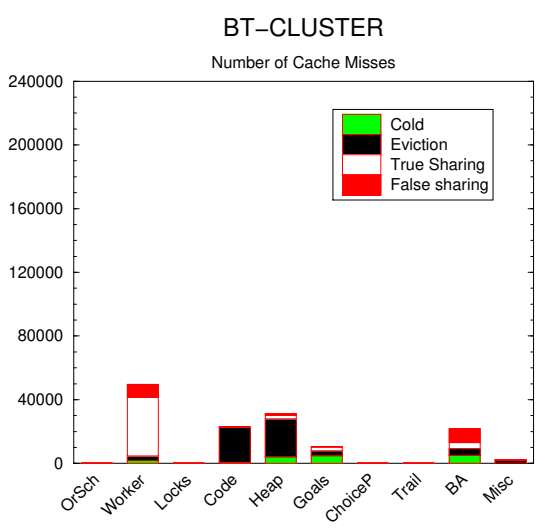


Figure 16: Misses by data area for `bt-cluster`

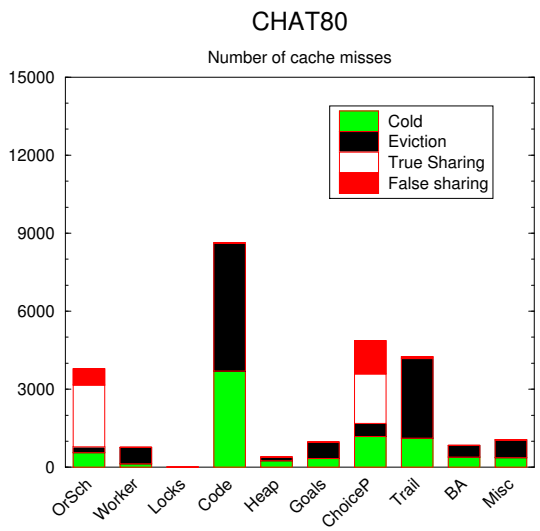


Figure 17: Misses by data area for `chat80`

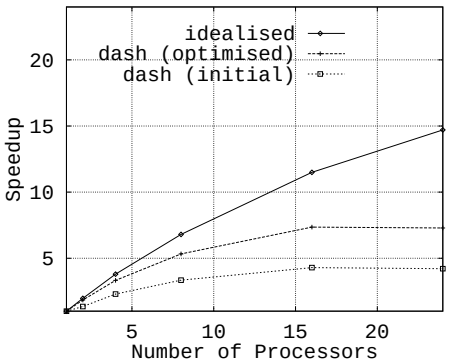


Figure 18: Speedups for `chat80`

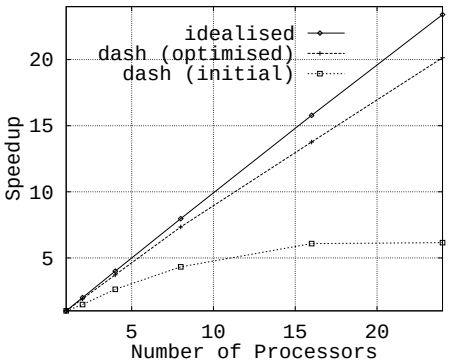


Figure 19: Speedups for `fp`

**And/Or-Parallel Benchmarks** Figure 20 shows the speedups of the new version of Andorra-I for the `pan2` benchmark. In this case, the speedup improvement resulting from our optimisations was quite small. The improvement for 24 workers was just of 13%, that is, from 4.8 to 5.4. This is about half of the speedup possible on the `idealised` machine.

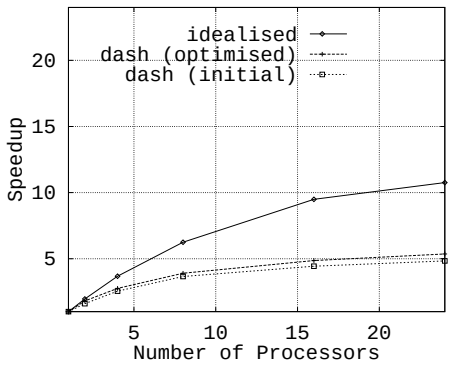


Figure 20: Speedups for `pan2`

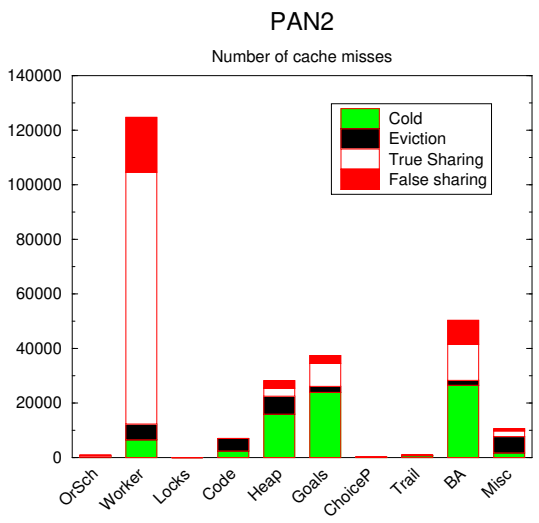


Figure 21: Misses by data area for `pan2`

Figure 21 shows the cache miss distribution for the optimised Andorra-I. Unsurprisingly, our optimisations had little impact on `pan2`, as the main source of misses was true sharing originating from the `Worker` region. A more detailed analysis proved that these misses originate from lack of work. Workers are searching each other's queues and generating misses. We are investigating more sophisticated scheduling strategies to address this problem.

# 6 Related Work

Initial studies of PLP systems were interested in designing novel architectures for parallel logic programming. Tick and Hermenegildo [21] give an interesting study of caching behaviour for independent and-parallelism. ICOT researchers studied the performance of a stream and-parallel system on a cluster-based architecture, the PIM machine [20], which was specifically built for logic programming.

In our study we are interested in general-purpose scalable architectures. Silva studied one or-parallel system, the DorPP, using a simulator of an early distributed shared-memory machine, the EDS [19], where data coherence was user controlled. He presents good speedups but for a limited number of processors. Parallel logic programming was one of the major targets for the DDM [16], one of the earliest COMA machines. Raina showed good scalability for highly-parallel benchmarks under the or-parallel system Aurora, a precursor of Andorra-I.

Our studies sought to determine the behaviour of the data structures in the parallel system, and pinpoint the main sources of cache misses. An alternative to this data-centric approach is the control-centric approach, where one studies how each line of the program contributes to the total number of misses. Recently, Montelius used SimICS, a simulator of SUN bus-based parallel machines, to study the AKL And/Or parallel system [15]. He detected that most misses originated from one line in the garbage collector, and was able to improve AKL's performance. We have also used SimICS to study our modified Andorra-I system. Our experience is that the control-centric approach is very useful

when the system has only a few line of code that cause problems. This type of strategy is not very helpful when the sources of misses are numerous and widely distributed within the system, just as in the original Andorra-I system.

# 7 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 scalable shared memory systems. The key to these results was the extensive data available from detailed simulations of Andorra-I. This information showed that there was no need to restructure the system or its schedulers. Instead, performance could be dramatically improved by focusing on accesses to shared data.

We have obtained very good results both for and-parallel and or-parallel benchmarks. Or-parallelism is a promising source of parallelism for large systems. Or-parallel applications tend to have restricted memory footprints, as most alternatives are still searched through backtracking. These applications often search solutions for combinatorial problems, which can grow to form very large search spaces. We are therefore not surprised that our best benchmark is an or-parallel benchmark.

We were positively surprised by the modified Andorra-I's good performance on the and-parallel benchmarks. In tasks such as memory management, Andorra-I requires more synchronisation than traditional and-parallel systems, and we would expect this to affect performance. Instead, the system obtained close to linear speedups for both our and-parallel benchmarks. Differently from or-parallel applications, the memory footprint of and-parallel benchmarks grows fast with problem size, and garbage collection is required for very large problems. We expect that this may limit speedups for larger numbers of processors.

We therefore believe there is potential for improving the performance of PLP systems even further. To prove so will require more radical changes to data structures within Andorra-I itself, as the system was simply not designed for such large numbers of processors. We are also interested in studying how to improve performance for benchmarks with less parallelism, such as `pan2`. We believe doing so will mainly require reducing the amount of busy-waiting in the system. Last, but not least, we are interested in studying performance of other parallel logic programming systems, such as the systems that exploit independent and-parallelism.

# 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. Our work has benefited from Márcio da Silva's studies of the sensitivity of Andorra-I to architectural parameters. Vítor Santos Costa would like to thank support from the PROLOPPE and MELODIA projects, and from the Fundação da Ciência e Tecnologia and PRAXIS programme.

# References

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

[2] 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.

[3] 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.

[4] 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.

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

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

[7] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, May 1993.

[8] Inês 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.

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

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

[11] László Béla Kovács. An Incremental Prolog Systems Development for Floor Plan Design by Dissecting. In *The Practical Application of Prolog, International Conference*, April 1992.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[14] 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.

[15] Johan Montelius and Peter Magnusson. Using SimICS to Evaluate the PennySystem. In *Proceedings of the 1994 International Logic Programming Symposium*, October 1997.

[16] 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.

[17] 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.

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

[19] Fernando M. A. Silva. Initial performance of dorpp: an or-parallel Prolog system for a distributed shared memory architecture. In *Proceedings of the ICLP'93 Postconference Workshop on Concurrent, Distributed and Parallel Implementation of Logic Programming Systems*, June 1993.

[20] Kazuo Taki. Parallel inference machine PIM. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 50–72. ICOT, 1992.

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.