ISRN SICS-R--92/07--SE

Performance of Muse on Switch-Based Multiprocessor Machines

by

Khayri A. M. Ali, Roland Karlsson and
Shyam Mudambi

SICS research report
R92:07
ISSN 0283-3638

Performance of Muse on Switch-Based Multiprocessor Machines

March 1992

Khayri A. M. Ali   Roland Karlsson   Shyam Mudambi

SICS, Swedish Institute of Computer Science
PO Box 1263
S-164 28 Kista, Sweden

THE Muse (multiple sequential Prolog engines) approach has been used to make a simple and efficient OR-parallel implementation of the full Prolog language. The performance results of the Muse system on bus-based multiprocessor machines have been presented in previous papers. This paper discusses the implementation and performance results of the Muse system on switch-based multiprocessors (the BBN Butterfly GP1000 and TC2000). The results of Muse execution show that high real speedups can be achieved for Prolog programs that exhibit coarse-grained parallelism. The scheduling overhead is equivalent to around 8 – 26 Prolog procedure calls per task on the TC2000. The paper also compares the Muse results with corresponding results for the Aurora OR-parallel Prolog system. For a large set of benchmarks, the results are in favor of the Muse system.

Contents

1 Introduction 1

2 The Muse Approach — An Overview 1

3 Muse on the Butterfly 3

4 Performance Results 5
4.1 Timings and Speedups 6
4.1.1 Butterfly GP1000 6
4.1.2 Butterfly TC2000 7
4.2 Worker Activities and Scheduling Overhead 12

5 Comparison with Aurora 16

6 Conclusions 20

7 Acknowledgments 22

1 Introduction

With the availability of multiprocessors many approaches to parallel implementations of the Prolog language have been developed. Some of them exploit either OR-parallelism [2, 5, 7, 10, 16, 18, 21, 27] or independent AND-parallelism [12, 15, 19] or a combination of both [6, 11, 17, 26]. The Muse approach [2] exploits OR-parallelism only. In OR-parallel execution branches of a Prolog search tree are executed in parallel.

Muse based on SICStus Prolog [9] (a fast and portable Prolog system) is currently implemented on both bus-based and switch-based shared memory machines. The bus-based machines are TP881V (a 4 (88100) processor machine from Tadpole Technology), a machine constructed at SICS (a 7 (68020) processor VME-bus machine with local/shared memory), Sequent Symmetry S81 (a 26 (i386) processor machine from Sequent). The switch-based machines are BBN Butterfly I (GP1000) and II (TC2000), with 96 (68020) and 45 (88100) processors respectively. The overhead associated with this adaptation is very low in comparison with the other approaches. It is around 3% to 5% for the bus-based machines. The performance results of Muse on the BBN Butterfly machines will be presented in this paper. The Muse implementation and its performance results on the constructed prototype and the Sequent Symmetry machine have been presented in previous papers [2, 3, 4, 1].

The paper is organized as follows. Section 2 briefly describes the Muse approach. Section 3 discusses Muse implementation on the Butterfly machines. Section 4 presents performance results of the Muse implementation on the Butterfly machines. Section 5 compares the Muse results with the corresponding results for another OR-parallel Prolog system, named Aurora. Section 6 concludes the paper.

2 The Muse Approach — An Overview

This section briefly describes the Muse approach presented in [2, 3, 4, 1]. In the Muse approach (as in other OR-parallel Prolog approaches, e.g. Aurora [21] and PEPSys [5]), OR-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. The Muse execution model is based on having a number of sequential Prolog engines, each with its own local address space, and some global address space shared by all engines. Each sequential Prolog engine is a worker with its own stacks[1]. The stacks are not shared between workers. Thus,

[1]The assumed worker's stacks are: *a choicepoint stack, an environment stack, a term stack, and a trail.* The first two correspond to the WAM local stack and the second two correspond to the WAM heap and trail respectively [24].

each worker has bindings associated with its current branch in its own copy of the stacks.

This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the difference between the two states. This reduces copying overhead. Nodes are shared between workers to allow dynamic load balancing, which reduces the frequency of copying.

A node in a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared (private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each shared node is accessible only to workers within the subtree rooted at the node. Private nodes are only accessible to the worker that created them.

Each worker can be in either engine mode or scheduler mode. In engine mode, the worker works exactly like a sequential Prolog engine on private nodes, but is also able to respond to requests from other workers. In scheduler mode, the worker establishes the necessary coordination with other workers. The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead. The sources of overhead in the Muse model include (1) copying part of a worker state, (2) making local nodes shareable, and (3) grabbing a piece of work from a shared node. The Muse scheduling work strategies to minimize the overhead are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the amount of shared work between the workers and allows each worker to release work from the bottommost node in its branch (dispatching on the bottommost) by using backtracking with almost no overhead. Dispatching on the bottommost also entails less speculative work than dispatching on the topmost, where work is taken from the topmost live node in a branch (see [14]). (Speculative work is defined as work which is within the scope of a cut and therefore may never be done in sequential execution.)
- When a worker runs out of work from its branch it will try to share work with the nearest worker which has maximum load. The load is measured by the number of local unexplored alternatives, and nearness is determined by the positions of workers in the search tree. This strategy attempts to maximize the shared work and minimize sharing overhead.
- Workers which cannot find any work in the system will try to distribute themselves over the tree and stay at positions where sharing of new work is expected to have low overhead.

- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position in the tree before requesting sharing from the busy worker. This allows a busy worker to concentrate on its task[2] and to respond only to requests that have to be handled by it.

The current implementation of Muse supports the full Prolog language with its standard semantics. It also supports asynchronous (parallel) side effects and internal database predicates. In this implementation we have simple mechanisms for supporting cut and all standard Prolog side effect predicates (e.g. read, write, assert, retract, etc).

A simple way to guarantee the correct semantics of sequential side effects is to allow execution of such side effects only in the leftmost branch of the whole search tree. The current Muse implementation does not support suspension of branches. Hence a worker that executes a sequential side effect predicate in a branch that is not leftmost, will wait until that branch becomes leftmost in the entire tree. Similarly, for supporting the *findall* predicate, a worker that generates a findall solution will wait until its branch is leftmost in a proper subtree.

The effect of *cut* N in a branch is to prune all branches to the right of the branch in the subtree rooted at the node N. In the current implementation of cut, when a worker executing a cut is not leftmost in the relevant subtree, it will prune as much as it can and leave the pruning of the remaining branches to a worker to its left and then proceed with executing operations following the cut. When a worker detects that all left branches have failed, it will try to prune as much as it can until all branches within the scope of the cut have been pruned.

3 Muse on the Butterfly

The Butterfly GP1000 is a multiprocessor machine capable of supporting up to 128 processors. The GP1000 is made up of two subsystems, the processor nodes and the Butterfly switch, which connects all nodes. A processor node consists of an MC68020 microprocessor, 4 MBytes of memory and a Processor Node Controller (PNC) that manages all references. A non-local memory access across the switch takes about 5 times longer than local memory access (when there is no contention). The Butterfly switch is a multi-stage omega interconnection network. The switch on the GP1000 has a hardware supported block copy operation, which is useful when implementing the Muse incremental copying strategy. The peak bandwidth of the switch is 4 MBytes per second per switch path.

The Butterfly TC2000 is a similar but newer machine. The main difference is that the processors used in the TC2000 are the Motorola 88100s. They are an order of

[2] A task is a continuous piece of work executed by a worker.

magnitude faster than the MC68020 and have two 16 KBytes data and instruction caches. Thus in the TC2000 there is actually a three level memory hierarchy: cache memory, local memory and remote memory. Unfortunately no support is provided for cache coherence of shared data. Hence by default shared data are not cached on the TC2000. The peak bandwidth of the Butterfly switch on the TC2000 is 9.5 times faster than the Butterfly GP1000 (around 38 MBytes per second per path).

The main optimization of Muse for the Butterfly machines was the creation of separate copies of the WAM code for each processor. The copies of the code were placed in non-shared memory, thus making them cachable on the TC2000. We have also tried to optimize the scheduler by reducing non-local busy waits and by optimal placement of memory. All the shared memory used is spread across all the nodes to avoid switch contention. We also identified and removed some hot spots. For example, data associated with a shared choicepoint can be simultaneously accessed by many workers (when looking for work), hence on the Butterfly machines access to this data was serialized. Similarly data which are most frequently accessed by a single worker (such as the global registers associated with each worker) are stored in its local memory.

To reduce copying overheads, the local address space of each worker is mapped into a separate part of the global address space of the system. This enables the two workers involved in copying to copy parts of the WAM stacks in parallel. A cache coherence protocol for the WAM stack areas has also been implemented on the TC2000. The basic idea of this protocol is as follows. Every worker keeps a list of all stack areas that it reads from any other worker during its last copying operation. When a worker Q is going to copy data from another worker P in the next copying operation, Q invalidates the listed areas in its cache and P flushes the areas to be copied by Q from its cache.

In the current Muse implementation, there is one cell associated with every choicepoint frame for supporting leftmost operations. These cells are accessible by all the workers. To simplify caching of the WAM stacks, these cells are saved in a separate stack associated with every worker. This stack is not cachable.

Another optimization of the Butterfly Muse implementation is in the installation of bindings. In the Muse model, bindings made by P in its stack parts common with Q need to be installed in Q's stacks. Since block copying is more efficient over the Butterfly switch, instead of installing cell by cell from P to Q, P first assembles those bindings, then Q copies them as a block into its local memory and installs the bindings.

4 Performance Results

In this section we discuss the performance of the Muse implementation on the Butterfly machines for a large group of benchmarks. The group of benchmarks used in this paper is the same group of benchmarks that has already been used for the evaluation of OR-parallel Prolog systems on bus-based multiprocessor architectures [2, 5, 13, 8, 23]. We extend this group with some benchmarks that have been used for the Aurora OR-parallel Prolog system on the Butterfly machines [22]. The group of benchmarks used in this paper can be divided into two sets: the first set (*11-queens1, 11-queens2, semigroup, 8-queens1, 8-queens2, tina, salt-mustard, parse2, parse4, parse5, db4, db5, house, parse1, parse3, farmer*) does not contain major cuts. This set contains benchmarks with different characteristics. It is divided into four groups which reflect the amount of parallelism exhibited by the benchmarks: (*11-queens1, 11-queens2, semigroup*), (*8-queens1, 8-queens2, tina, salt-mustard*), (*parse2, parse4, parse5, db4, db5, house*), and (*parse1, parse3, farmer*). The four groups are referred to in the following sections as *GI, GII, GIII*, and *GIV* respectively. The group *GI* represents programs with the highest amount of parallelism, and the group *GIV* represents programs with the lowest amount of parallelism.

N-queens1 and *N-queens2* are two different N queens programs from ECRC. *semigroup* is a theorem proving program for studying the R-classes of a large semigroup from Argonne National Laboratory [20]. *tina* is a holiday planning program from ECRC. *salt-mustard* is the "salt and mustard" puzzle from Argonne. *parse1 – parse5* are queries to the natural language parsing parts of Chat-80 by F. C. N. Pereira and D. H. D. Warren. *db4* and *db5* are the database searching parts of the fourth and fifth Chat-80 queries. *house* is the "who owns the zebra" puzzle from ECRC. *farmer* is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC. All benchmarks of the first set look for all solutions of the problem.

The benchmarks in the second set (*mm1, mm2, mm3, mm4, num1, num2, num3, num4*) contain major cuts and have been used for studying different cut schemes [13]. *mm* is a master mind program with four different secret codes. The numbers program (*num*) generates the two largest numbers consisting of given digits and fulfilling specified requirements. The numbers program was run with four different queries. This set is divided into two groups known in the following sections as *mm* and *num*. All benchmarks of the second set look for the first solution of the problem.

In this section we present the Muse results for the first set of benchmarks. The Muse results for the second set of benchmarks are presented in Section 5.

4.1 Timings and Speedups

4.1.1 Butterfly GP1000

Table 1 shows the execution times (in seconds) from the execution of the first set of benchmarks for Muse on the GP1000 machine. The execution times given in this paper are the mean values obtained from eight runs. On the Butterfly machines, mean values are more reliable than best values due to variations of timing results from one run to another. These variations are due mainly to switch contention and are greatest in the smaller benchmarks.

Benchmarks	Muse Workers 1	10	30	50	64	SICStus0.6
semigroup	7948.49	797.91(9.96)	271.54(29.3)	168.08(47.3)	134.55(59.1)	7363.96(1.08)
11-queens1	1706.24	171.23(9.96)	57.99(29.4)	35.93(47.5)	28.54(59.8)	1595.00(1.07)
11-queens2	5195.67	521.06(9.97)	175.47(29.6))	108.36(47.9)	86.93(59.8)	4829.65(1.08)
• GI Σ	14850.40	1490.07(9.97)	504.69(29.4)	312.25(47.6)	249.81(59.4)	13788.61(1.08)
8-queens1	13.25	1.49(8.89)	0.74(17.9)	0.71(18.7)	0.82(16.2)	12.36(1.07)
8-queens2	33.00	3.67(8.99)	1.53(21.6)	1.27(26.0)	1.40(23.6)	31.66(1.04)
tina	27.28	3.38(8.07)	1.83(14.9)	1.70(16.0)	1.80(15.2)	23.90(1.14)
salt-mustard	4.29	0.54(7.94)	0.38(11.3)	0.43(9.98)	0.51(8.41)	3.44(1.25)
○ GII Σ	77.82	9.07(8.58)	4.46(17.4)	4.10(19.0)	4.47(17.4)	71.36(1.09)
parse2*20	10.27	3.27(3.14)	3.55(2.89)	3.63(2.83)	3.78(2.72)	9.38(1.09)
parse4*5	9.42	1.91(4.93)	1.83(5.15)	1.88(5.01)	1.91(4.93)	8.67(1.09)
parse5	6.65	1.05(6.33)	0.92(7.23)	1.00(6.65)	1.02(6.52)	6.10(1.09)
db4*10	4.24	0.98(4.33)	1.07(3.96)	1.13(3.75)	1.14(3.72)	3.80(1.12)
db5*10	5.16	1.20(4.30)	1.25(4.13)	1.32(3.91)	1.40(3.69)	4.64(1.11)
house*20	6.94	2.92(2.38)	3.19(2.18)	3.37(2.06)	3.43(2.02)	8.86(1.01)
◇ GIII Σ	42.68	11.30(3.78)	11.80(3.62)	12.31(3.47)	12.61(3.38)	39.53(1.08)
parse1*20	2.72	1.63(1.67)	1.73(1.57)	1.80(1.51)	2.03(1.34)	2.48(1.10)
parse3*20	2.33	1.60(1.46)	1.64(1.42)	1.90(1.23)	1.78(1.31)	2.13(1.09)
farmer*100	6.02	4.82(1.25)	4.86(1.24)	5.30(1.14)	5.26(1.14)	5.58(1.08)
* GIV Σ	11.07	8.04(1.38)	8.22(1.35)	8.82(1.26)	9.03(1.23)	10.19(1.09)
Σ	14981.97	1518.51(9.87)	529.21(28.3)	337.68(44.4)	276.01(54.3)	13909.69(1.08)

Table 1: Muse execution times (in seconds) for the first set of benchmarks on the GP1000.

In Table 1, times are shown for 1, 10, 30, 50 and 64 workers with speedups given in parentheses. These speedups are relative to the execution times of Muse on one worker. The last column shows execution times of SICStus0.6 on one GP1000 node with the ratio of execution times on Muse for the 1 worker case to the SICStus0.6 times. For benchmarks with small execution times the timings shown refer to repeated runs, the repetition factor being shown in the first column. $\sum$ in the last row corresponds to the goal: *(11-queens1, 11-queens2, semigroup, 8-queens1, 8-queens2, tina, salt-mustard, parse2*20, parse4*5, parse5, db4*10, db5*10, house*20, parse1*20, parse3*20, farmer*100)*. That is, the timings shown in the last row correspond to running the whole first set of benchmarks as one benchmark. In all tables, the last row for each group of a set of benchmarks represents the whole group as one benchmark.

As shown in the last column of Table 1, on one worker, Muse is about 8% slower

than SICStus0.6, the sequential Prolog system from which Muse is derived. This overhead is mainly due to the maintaining of the private load and the checking of some global flags by each Muse worker. This overhead is higher for programs that access the Prolog tables[3] heavily. The Prolog tables are partitioned into sections and each section resides in the local memory of one processor. Accessing remote memory is much more expensive than accessing local memory. The ratio of local to remote memory access time on the GP1000 is 1 to 5. The *salt-mustard* benchmark has 25% overhead per worker, because it makes heavy use of meta calls, which require accessing the predicate table. The corresponding overhead in the other OR-parallel Prolog systems is much higher (see Section 5).

The performance results that Table 1 illustrates are encouraging: on 64 processors the average speedup factor is 59.4 for the *GI* programs, 17.4 for the group *GII*, 3.38 for the group *GIII*, and around 1.23 for the group *GIV*. The speedup factor for the entire first set of benchmarks on 64 processors is 54.3. The average real speedups on 64 processors, in comparison to SICStus on one GP1000 processor, are 55.2 for the *GI* programs, and 50.4 for the entire first set of benchmarks. For all programs in the group *GI*, increasing the number of workers results in shorter execution times. For programs in the groups *GII, GIII* and *GIV*, increasing the number of workers beyond a certain limit results in no further improvement of execution times. Actually, increasing the number of workers results in slightly longer execution times for the latter three groups. This degradation is due to the increased scheduling overhead (see Section 4.2).

Figure 1 shows the speedup curves for the four groups: *GI, GII, GIII* and *GIV*. These curves correspond to speedups obtained from Table 1. Variations around the mean value are shown by a vertical line with two short horizontal lines at each end. Variations of less than 0.4 are not shown (in all figures).

We observe from Figure 1 that when the amount of parallelism is not enough for all workers in the system to find work, the speedup curves level off and reach an almost constant value. This characteristic is very important for any parallel system (such as Muse) that dynamically schedules work at runtime.

4.1.2 Butterfly TC2000

On the TC2000 we have two versions of the Muse system: one caches the WAM stacks and the other does not. The version that caches the WAM stacks has a faster Prolog engine. The timing results on the two versions will be presented in this section to illustrate the effect of engine speed on the obtained speedups.

Tables 2 and 3 present the execution times of Muse for the first set of benchmarks on

[3]By Prolog tables we mean the atom table, the predicate table, and also the allocated records associated with occupied items in the tables.

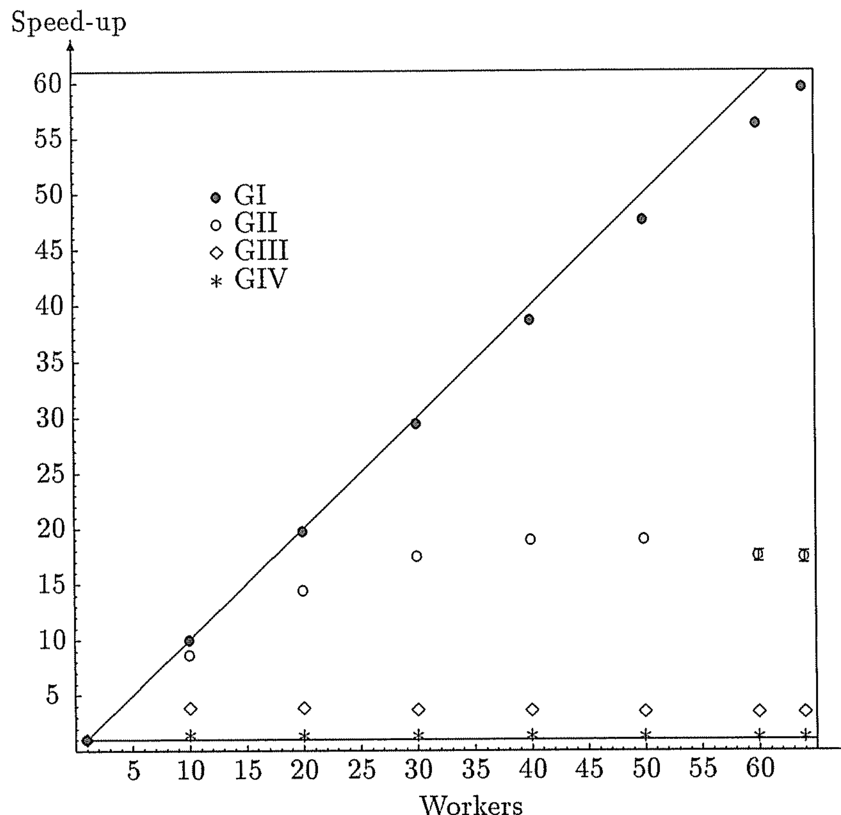

Figure 1: Speedups of Muse for the first set of benchmarks on GP1000.

the TC2000 machine when the WAM stacks were and were not cached respectively. Times are shown for 1, 10, 20, 30 and 32 workers with speedups given in parentheses. These speedups are relative to the execution times of Muse on one worker as in Table 1. Figures 2 and 3 show speedup curves corresponding to Tables 2 and 3.

From results shown in Tables 2 and 3, we observe that the absolute times on the Muse version that cached the WAM stacks are shorter than the corresponding ones on the version that did not. We also observe that speedups in Table 2 are somewhat better than the corresponding ones in Table 3. The reason speedups are poorer in Table 3 is that the Muse version that caches the WAM stacks has a faster Prolog engine, and in general when the ratio of the speed of the engine to that of the scheduler increases, the relative cost of scheduling overhead increases, which in turn decreases the speedups.

From the last column in Table 2, Muse on one worker is about 59% slower than SICStus0.6. This is because in the SICStus system all code, data, and tables are cachable while in this version of Muse only the code area is cachable. (The TC2000 processor is provided with data and instruction caches.) The overhead per Muse worker has been reduced from 59% to 22% by caching the data area (i.e. the WAM

Benchmarks	Muse Workers 1	10	20	30	32	SICStus0.6
semigroup	1502.60	151.18(9.94)	76.44(19.7)	51.62(29.1)	48.47(31.0)	990.89(1.52)
11-queens1	313.89	31.66(9.91)	16.00(19.6)	10.80(29.1)	10.11(31.0)	190.87(1.64)
11-queens2	965.24	96.68(9.98)	48.67(19.8)	32.71(29.5)	30.75(31.4)	574.27(1.68)
• GI Σ	2781.73	279.50(9.95)	141.10(19.7)	95.11(29.2)	89.32(31.1)	1756.03(1.58)
8-queens1	2.47	0.29(8.52)	0.19(13.0)	0.15(16.5)	0.15(16.5)	1.48(1.67)
8-queens2	6.36	0.69(9.22)	0.39(16.3)	0.30(21.2)	0.30(21.2)	3.77(1.69)
tina	5.57	0.71(7.85)	0.45(12.4)	0.39(14.3)	0.39(14.3)	3.07(1.81)
salt-mustard	0.84	0.11(7.64)	0.09(9.33)	0.09(9.33)	0.09(9.33)	0.44(1.91)
∘ GII Σ	15.24	1.79(8.51)	1.10(13.9)	0.94(16.2)	0.93(16.4)	8.76(1.74)
parse2*20	2.36	0.84(2.81)	0.93(2.54)	0.93(2.54)	0.94(2.51)	1.43(1.65)
parse4*5	2.16	0.49(4.41)	0.47(4.60)	0.52(4.15)	0.53(4.08)	1.31(1.65)
parse5	1.53	0.29(5.28)	0.26(5.88)	0.26(5.88)	0.27(5.67)	0.93(1.65)
db4*10	0.90	0.21(4.29)	0.22(4.09)	0.23(3.91)	0.23(3.91)	0.53(1.70)
db5*10	1.09	0.25(4.36)	0.26(4.19)	0.26(4.19)	0.26(4.19)	0.64(1.70)
house*20	1.33	0.62(2.15)	0.66(2.02)	0.66(2.02)	0.68(1.96)	0.94(1.41)
⋄ GIII Σ	9.37	2.70(3.47)	2.80(3.35)	2.85(3.29)	2.89(3.24)	5.78(1.62)
parse1*20	0.63	0.44(1.43)	0.45(1.40)	0.47(1.34)	0.46(1.37)	0.38(1.66)
parse3*20	0.54	0.43(1.26)	0.44(1.23)	0.45(1.20)	0.46(1.17)	0.33(1.64)
farmer*100	1.13	1.04(1.09)	1.05(1.08)	1.07(1.06)	1.11(1.02)	0.66(1.71)
* GIV Σ	2.30	1.90(1.21)	1.93(1.19)	1.98(1.16)	2.02(1.14)	1.37(1.68)
Σ	2808.64	285.89(9.82)	146.93(19.1)	100.89(27.8)	95.18(29.5)	1771.94(1.59)

Table 2: Muse execution times (in seconds) for the first set of benchmarks on TC2000 without caching of the WAM stacks.

Benchmarks	Muse Workers 1	10	20	30	32	SICStus0.6
semigroup	1178.63	118.56(9.94)	60.06(19.6)	40.72(28.9)	38.35(30.7)	990.89(1.19)
11-queens1	225.23	22.78(9.89)	11.58(19.4)	7.88(28.6)	7.41(30.4)	190.87(1.18)
11-queens2	729.77	73.12(9.98)	36.92(19.8)	24.93(29.3)	23.43(31.1)	574.27(1.27)
• GI Σ	2133.63	214.46(9.95)	108.53(19.7)	73.52(29.0)	69.18(30.8)	1756.03(1.22)
8-queens1	1.79	0.21(8.52)	0.14(12.8)	0.13(13.8)	0.13(13.8)	1.48(1.21)
8-queens2	4.80	0.53(9.06)	0.31(15.5)	0.26(18.5)	0.25(19.2)	3.77(1.27)
tina	4.28	0.58(7.38)	0.39(11.0)	0.34(12.6)	0.34(12.6)	3.07(1.39)
salt-mustard	0.71	0.10(7.10)	0.08(8.88)	0.09(7.89)	0.10(7.10)	0.44(1.61)
∘ GII Σ	11.58	1.42(8.15)	0.91(12.7)	0.82(14.1)	0.81(14.3)	8.76(1.32)
parse2*20	1.82	0.80(2.27)	0.84(2.17)	0.85(2.14)	0.87(2.09)	1.43(1.27)
parse4*5	1.67	0.42(3.98)	0.43(3.88)	0.45(3.71)	0.47(3.55)	1.31(1.27)
parse5	1.18	0.24(4.92)	0.22(5.36)	0.24(4.92)	0.24(4.92)	0.93(1.27)
db4*10	0.68	0.20(3.40)	0.21(3.24)	0.21(3.24)	0.21(3.24)	0.53(1.28)
db5*10	0.83	0.23(3.61)	0.24(3.46)	0.25(3.32)	0.25(3.32)	0.64(1.30)
house*20	0.98	0.58(1.69)	0.61(1.61)	0.63(1.56)	0.63(1.56)	0.94(1.04)
⋄ GIII Σ	7.16	2.46(2.91)	2.54(2.82)	2.62(2.73)	2.65(2.70)	5.78(1.24)
parse1*20	0.48	0.40(1.20)	0.40(1.20)	0.41(1.17)	0.42(1.14)	0.38(1.26)
parse3*20	0.41	0.39(1.05)	0.41(1.00)	0.42(0.98)	0.42(0.98)	0.33(1.24)
farmer*100	0.83	1.01(0.82)	1.03(0.81)	1.07(0.78)	1.05(0.79)	0.66(1.26)
* GIV Σ	1.72	1.80(0.96)	1.83(0.94)	1.89(0.91)	1.88(0.91)	1.37(1.26)
Σ	2154.09	220.14(9.79)	113.83(18.9)	78.85(27.3)	74.53(28.9)	1771.94(1.22)

Table 3: Muse execution times (in seconds) for the first set of benchmarks on TC2000 with caching of the WAM stacks.

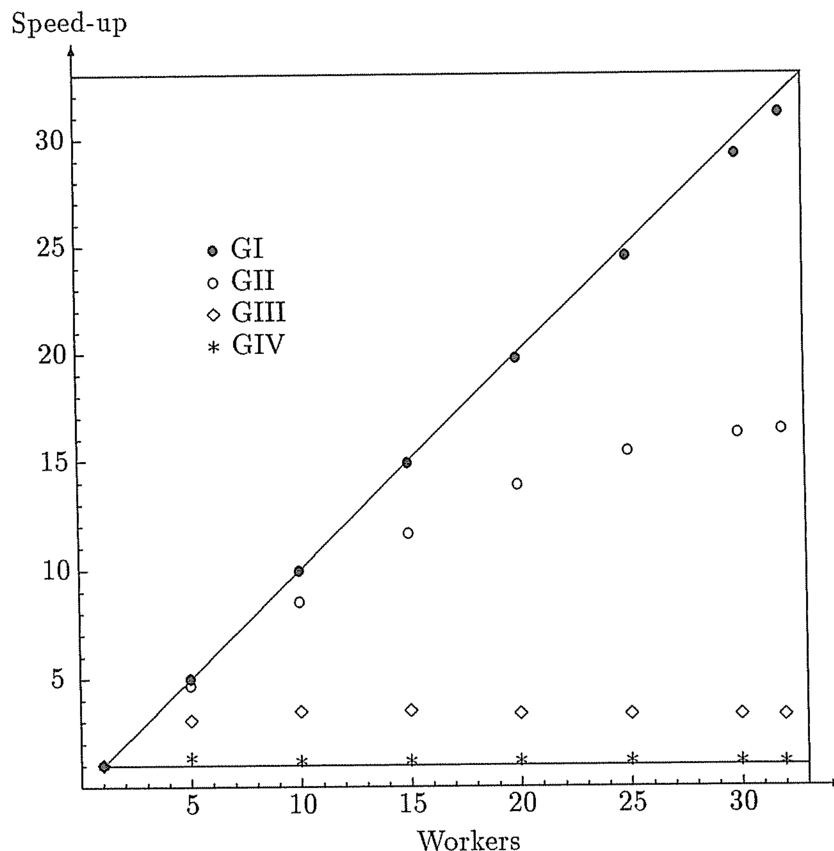

Figure 2: Speedups of Muse for the first set of benchmarks on TC2000 without caching of the WAM stacks.

stacks) as shown in the last column in Table 3. The Prolog tables on Muse system are not cachable. The reason the overhead per Muse worker is lower on the GP1000 (8%) than on TC2000 (22%) is that the former does not have any cache memory. So, on GP1000 there is no difference in access time between cachable and noncachable data while on TC2000 there is a difference. In order to verify that it was the Prolog tables that caused the overhead, we compared the execution times of Muse with one worker (making the Prolog tables cachable) to those of SICStus0.6 on the TC2000 (Table 4). We see that the overhead for this version of Muse is between 4% to 7%, which is quite close to the overhead observed on the Sequent Symmetry.

The average real speedups on 32 processors, in comparison to SICStus on one TC2000 processor, are 25.4 for the *GI* programs, and 23.8 for the whole first set of benchmarks (calculated from Table 3). It should be noted that the Muse system has always been shown to have the best real speedups in comparison to the other existing OR-Parallel Prolog systems (see Section 5 for further discussions).

The role of the ratio of the speed of the engine to that of the scheduler in the Muse system on the Butterfly machines can also be seen in the speedup curves those of

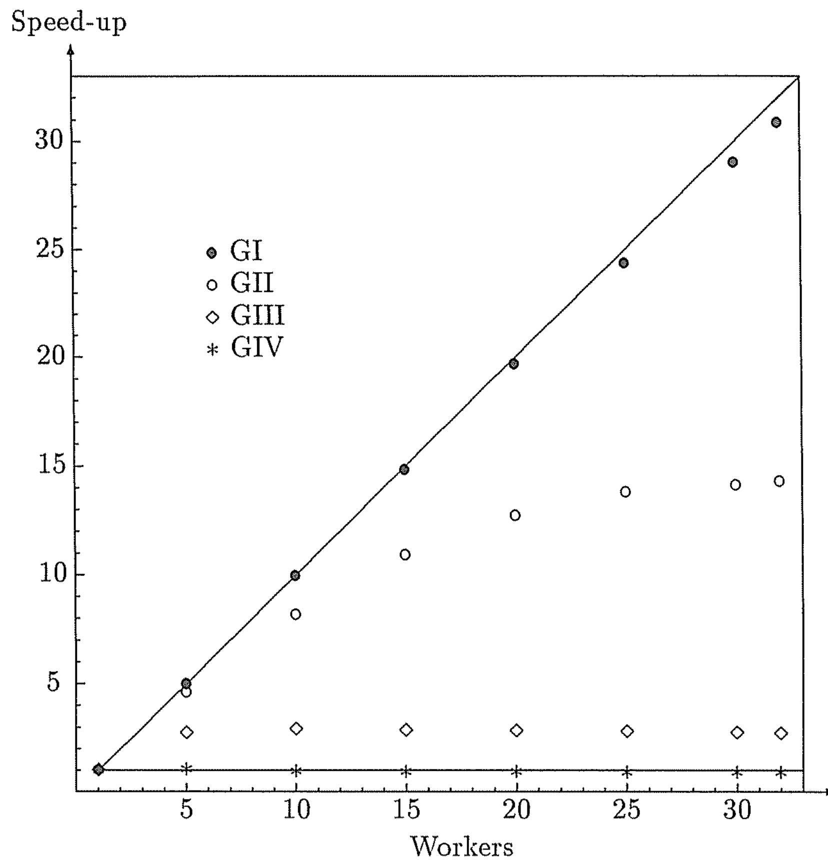

Figure 3: Speedups of Muse for the first set of benchmarks on TC2000 with caching of the WAM stacks.

Benchmarks	Muse	SICStus0.6
GI	1829.03	1756.03(1.04)
GII	9.36	8.76(1.07)
GIII	6.19	5.78(1.07)
GIV	1.47	1.37(1.07)
Σ	1846.05	1771.94(1.04)

Table 4: Comparison of the execution times (in seconds) between 1 Muse worker (with Prolog tables cached) and SICStus0.6.

Figures 1, 2 and 3. The best speedup curves are those of Figure 1 (for the same number of workers), and the worst curves are in Figure 3. From the first column in Tables 1 and 3, the engine speed on the TC2000 is around 7 times the engine speed on the GP1000 due to the difference in the speeds of the processors. The scheduler speed on the TC2000 is not increased by the same factor even though the switch speed is increased by a similar factor. There could be two reasons for this:

1. The scheduler data on the TC2000 are not cached (for obvious reasons), whereas most of the engine data are cached. This results in the engine to scheduler speed ratio being higher on the TC2000, as the GP1000 does not have a cache.
2. Unlike the GP1000 switch, the switch on the TC2000 does not have a hardware supported block copy operation. This has the effect of reducing the real bandwidth of the switch in incremental copying.

4.2 Worker Activities and Scheduling Overhead

In this section, we present and discuss the time spent in the basic activities of a Muse worker on the TC2000 machine for the version of Muse that caches the WAM stacks. A worker's time is distributed over the following activities:

1. *Prolog:* time spent executing Prolog, checking for the arrival of requests, and keeping the value of the private load up to date.
2. *Idle:* time spent waiting for work to be generated when there is temporarily no work available in the system.
3. *Grabbing Work:* time spent grabbing available work from shared nodes.
4. *Sharing:* time spent making private nodes shared with other workers, copying parts of the WAM stacks, binding installation, or synchronization with other workers while performing the sharing operation.
5. *Looking for work:* time spent looking for a worker with private load.
6. *Others:* time spent in other activities such as acquiring spin locks, sending requests to other workers – either requesting sharing or performing commit/cut to a shared node, etc.

Table 5 shows the time spent in each activity as a percentage of the total time, for one selected benchmark from each group of the first set of benchmarks. The last row of each table gives the scheduling overhead, which is the sum of all the activities excluding *Prolog* and *Idle*. The *11-queens2* is selected from *GI*, *8-queens2* from

GII, *parse5* from *GIII*, and *farmer* from *GIV*. Results shown in Table 5 have been obtained from an instrumented system of Muse on the TC2000. The times obtained from an instrumented system are longer than those obtained from an uninstrumented system by around 13%. We believe that the percentage of time spent in each activity obtained from the instrumented system reflects what is happening in the uninstrumented system.

In all benchmarks, the *Prolog* percentage of time decreases and the *Idle* percentage of time increases as the number of workers is increased. This is because each benchmark allows a limited amount of parallelism and increasing the number of workers decreases the amount of work assigned to each worker. For *11-queens2*, a program with high OR-parallelism, the *Prolog* percentage decreases by 1.1% from 3 workers to 32 workers and the *Idle* percentage increases by 0.4% from 3 workers to 32 workers. The total percentage of scheduling overhead (activities 3 – 6) on 32 workers is only 0.7%.

For *8-queens2*, the scheduling overhead increases from 0.9% on 3 workers to 17% on 32 workers. The rate by which the *Prolog* percentage decreases and the *Idle* percentage increases (w.r.t. to the number of workers) is much higher in *8-queens2* than in *11-queens2*. This is due to the fact that *8-queens2* has much less parallelism.

For *parse5* on 32 workers, the *Prolog* percentage is only 20.8%, while the *Idle* percentage reaches 52.4%. This benchmark has a reasonable amount of parallelism up to 10 – 15 workers, but not beyond. The scheduling overhead is higher than in the previous two benchmarks (5.3% on 3 workers and around 26% on 10 or more workers). *parse5* contains finer grain parallelism than in *8-queens2* and *11-queens2*.

For *farmer*, the amount of parallelism is only enough for 2 – 3 workers. The scheduling overhead reaches its maximum value (33.9%) for 3 workers and then decreases with an increasing number of workers. The reason for the decrease in the percentage of scheduling overhead is that adding more than 3 workers just increases the total *Idle* time in the system, which then dominates the total execution time.

In all benchmarks, the sharing overhead is the dominating part of the scheduling overhead. Sharing overhead ranges from 0.0% on the *11-queens2* to 22.3% on the *parse5*.

A possible explanation for the increase in overhead with the number of workers is shown in Table 6, which shows the effect of increasing the number of workers on the average task sizes (expressed as the number of Prolog procedure calls per task). A task is a continuous piece of work executed by a worker.

In Table 6 the task size decreases as the the number of workers is increased until it reaches a constant value, 20 for *parse5* and 11 for *farmer*. The reason the task size is almost constant after a certain number of workers is that the Muse system supports a form of delayed release of work that tries to avoid a continuous decrease in task size

	Muse Workers								
Activity	1	3	5	10	15	20	25	30	32
11-queens2									
Prolog	100	100	99.9	99.8	99.7	99.5	99.4	98.9	98.9
Idle	0	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.4
Grabbing Work	0	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.3
Sharing Work	0	0.0	0.0	0.0	0.1	0.2	0.2	0.3	0.3
Looking for Work	0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1
Others	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Sched. Overhead	0	0.0	0.0	0.1	0.2	0.4	0.4	0.7	0.7
8-queens2									
Prolog	100	98.9	97.6	93.0	87.5	81.3	74.0	67.0	64.3
Idle	0	0.2	0.5	2.0	4.2	8.1	11.5	16.6	18.7
Grabbing Work	0	0.3	0.7	1.8	3.0	3.6	5.0	5.5	5.2
Sharing Work	0	0.5	0.9	2.3	3.9	5.1	6.3	7.7	8.0
Looking for Work	0	0.1	0.2	0.7	1.2	1.6	2.7	2.6	3.1
Others	0	0.0	0.0	0.2	0.3	0.3	0.4	0.7	0.7
Sched. Overhead	0	0.9	1.8	5.0	8.4	10.6	14.4	16.5	17.0
parse5									
Prolog	100	93.9	84.6	60.6	46.5	34.0	27.5	22.4	20.8
Idle	0	0.8	2.1	13.3	21.3	30.5	41.0	48.4	52.4
Grabbing Work	0	1.8	2.9	3.4	3.9	3.9	3.5	3.3	3.0
Sharing Work	0	3.1	8.3	16.8	21.0	22.3	19.7	18.1	16.8
Looking for Work	0	0.4	2.0	5.7	6.9	8.7	7.8	7.2	6.4
Others	0	0.0	0.1	0.2	0.3	0.5	0.6	0.6	0.6
Sched. Overhead	0	5.3	13.3	26.1	32.1	35.4	31.6	29.2	26.8
farmer									
Prolog	100	44.8	25.7	12.0	7.8	5.8	4.6	3.8	3.5
Idle	0	21.3	42.6	70.3	80.6	85.8	88.7	90.6	91.2
Grabbing Work	0	5.0	4.1	2.1	1.4	1.0	0.8	0.7	0.7
Sharing Work	0	21.5	20.6	11.4	7.5	5.4	4.3	3.7	3.5
Looking for Work	0	6.0	6.1	3.6	2.4	1.7	1.4	1.1	1.1
Others	0	1.3	1.0	0.6	0.4	0.2	0.2	0.2	0.2
Sched. Overhead	0	33.8	31.8	17.7	11.7	8.3	6.7	5.7	5.5

Table 5: Percentage of time spent in basic activities of a Muse worker on TC2000.

Benchmark	Muse Workers 1	3	5	10	15	20	25	30	32
11-queens2	31410506	52004	20530	8412	3839	2712	2267	1463	1382
8-queens2	207778	980	618	174	124	91	66	61	58
parse5	39119	132	58	43	28	23	20	20	20
farmer	33486	16	11	11	11	11	11	11	11

Table 6: Task sizes for some programs of the first set of benchmarks on TC2000.

as the number of workers is increased. Supporting such a mechanism is crucial in order to avoid a continuous high increase in scheduling overhead with an increasing number of workers. The idea of the delayed release mechanism supported in the Muse system is as follows. When a worker reaches a situation in which it has only one private parallel node, it will make its private load visible to the other workers only when that node is still alive after a certain number, k, of Prolog procedure calls. The value of k is a constant selected to be larger than the number of Prolog procedure calls equivalent to the scheduling overhead per task (see below).

Table 7 shows the scheduling overhead per task in terms of Prolog procedure calls for the four selected benchmarks. The scheduling overhead (all activities except *Prolog* and *Idle*) is equivalent to around 8 – 26 Prolog procedure calls per task. It is almost constant (around 10) for *11-queens2*, a program with coarse-grained parallelism. It is higher and increases with the number of workers for the other three, programs with finer grain parallelism. For *parse5*, the scheduling overhead per task is somewhat higher in comparison with the other benchmarks. *parse5* generates a search tree with two long branches and each branch has many short branches.

Benchmark	Muse Workers 1	3	5	10	15	20	25	30	32
11-queens2	0	10.8	10.2	9.0	9.2	9.8	10.2	10.7	11.4
8-queens2	0	9.0	11.9	9.3	11.8	11.9	12.9	14.9	15.4
parse5	0	7.5	9.1	18.5	19.3	24.0	22.9	26.0	25.8
farmer	0	12.1	13.6	16.2	16.4	16.0	16.2	16.5	17.0

Table 7: Scheduling overheads per task in terms of Prolog procedure calls for some programs of the first set of benchmarks on TC2000.

To conclude, the scheduling overhead per task of Muse for the selected four benchmarks on TC2000 when caching the WAM stacks is around 8 – 26 Prolog procedure calls per task. This value is affected by the relative speeds of the Prolog engine and the scheduler. For instance, the corresponding value for the Muse system on Sequent Symmetry is around 5 – 7 Prolog procedure calls per task. The time of a Prolog procedure call is between 23 and 30 microseconds on TC2000, and between 83 and 100 microseconds on Sequent Symmetry. So, in order to avoid losing any

gain obtained by exploiting parallelism, the Muse system on TC2000 should avoid letting the task sizes fall below 26 Prolog procedure calls. That is, the value of k should be larger than 26. Actually, all results presented in this paper correspond to k equal to 10.

We have also experimented with increasing k to 15, 25 and 30, the result being that some improvements were obtained for programs with very fine granularity (e.g. *farmer*).

5 Comparison with Aurora

In this section we compare the timing results of Muse with the corresponding results for Aurora with the Manchester scheduler [21]. Both Aurora and Muse are based on the same sequential Prolog, SICStus version 0.6, and have been implemented on the same Butterfly machines. Neither Muse nor Aurora with the Manchester scheduler [8] do any special treatment of speculative work. (A new Aurora scheduler which handles speculative work better is under development at the University of Bristol.) The main difference between Muse and Aurora in the implementation of cut, findall, and sequential side effect constructs is that Aurora supports suspension of branches whereas Muse does not. For instance, an Aurora worker executing a cut in a non–leftmost branch of the cut node will suspend the branch and try to find work outside the cut subtree. In Muse, the pruning operation suspends while the worker proceeds with the next operation following the cut as described in Section 2.

Another difference between Aurora and Muse is that Aurora is based on a different model for OR-parallel execution of Prolog, namely the SRI model [25]. The idea of the SRI model is to extend the conventional WAM with a binding array for each worker and modify the trail to contain address-value pairs instead of just addresses. Each array is used by just one worker to store and access conditional bindings. A binding is conditional if a variable can get several bindings. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around in the tree. When a worker finishes a task, it moves over the tree to take another task. The worker starting a new task must partially reconstruct its array using the trail of the worker from which the task is taken.

Many optimizations have been made of the implementation of Aurora on the Butterfly machines. These optimizations are the same as the Muse optimizations, with the exception of caching the WAM stacks. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. Therefore, it is straightforward in Muse to make the WAM stack areas cachable whereas in Aurora it requires a complex cache coherence protocol to achieve this effect.

The Manchester scheduler for Aurora also tries to avoid a continuous decrease in

task sizes with an increasing number of workers. The idea used by the Manchester scheduler is that each busy worker checks for the arrival of requests from other workers on every N Prolog procedure calls. The best value of N on TC2000 is 20. All Aurora timing results presented in this paper correspond to $N = 20$.

Table 8 shows the execution times of Aurora with the Manchester scheduler for the first set of benchmarks on TC2000. The last column of Table 8 shows the execution times of SICStus0.6 on one TC2000 node with the ratio of execution times on Muse for the 1 worker case to the SICStus0.6 times.

Benchmarks	Aurora Workers 1	10	20	30	32	SICStus0.6
semigroup	1699.78	169.62(10.0)	87.03(19.5)	58.90(28.9)	54.68(31.1)	990.89(1.72)
11-queens1	369.14	36.92(10.00)	18.54(19.9)	12.47(29.6)	11.70(31.6)	190.87(1.93)
11-queens2	1044.49	105.38(9.91)	52.83(19.8)	35.37(29.5)	33.19(31.5)	574.27(1.82)
• GI Σ	3113.41	311.91(9.98)	158.38(19.7)	106.68(29.2)	99.55(31.3)	1756.03(1.77)
8-queens1	2.85	0.33(8.64)	0.21(13.6)	0.22(13.0)	0.23(12.4)	1.48(1.93)
8-queens2	6.97	0.76(9.17)	0.43(16.2)	0.35(19.9)	0.36(19.4)	3.77(1.85)
tina	6.33	0.83(7.63)	0.65(9.74)	0.88(7.19)	1.16(5.46)	3.07(2.06)
salt-mustard	1.47	0.19(7.74)	0.12(12.2)	0.12(12.2)	0.14(10.5)	0.44(3.34)
∘ GII Σ	17.62	2.10(8.39)	1.40(12.6)	1.55(11.4)	1.81(9.73)	8.76(2.01)
parse2*20	2.46	1.30(1.89)	1.50(1.64)	1.74(1.41)	1.87(1.32)	1.43(1.72)
parse4*5	2.25	0.74(3.04)	0.81(2.78)	1.01(2.23)	1.08(2.08)	1.31(1.72)
parse5	1.59	0.46(3.46)	0.47(3.38)	0.54(2.94)	0.59(2.69)	0.93(1.71)
db4*10	0.91	0.28(3.25)	0.35(2.60)	0.55(1.65)	0.58(1.57)	0.53(1.72)
db5*10	1.11	0.30(3.70)	0.36(3.08)	0.54(2.06)	0.60(1.85)	0.64(1.73)
house*20	1.55	0.79(1.96)	1.12(1.38)	1.99(0.78)	2.63(0.59)	0.94(1.65)
⋄ GIII Σ	9.87	3.86(2.56)	4.60(2.15)	6.24(1.58)	7.17(1.38)	5.78(1.71)
parse1*20	0.66	0.62(1.06)	0.73(0.90)	0.85(0.78)	0.84(0.79)	0.38(1.74)
parse3*20	0.57	0.62(0.92)	0.71(0.80)	0.70(0.81)	0.74(0.77)	0.33(1.73)
farmer*100	1.14	1.80(0.63)	2.14(0.53)	2.33(0.49)	2.37(0.48)	0.66(1.73)
* GIV Σ	2.37	3.03(0.78)	3.56(0.67)	3.80(0.62)	3.91(0.61)	1.37(1.73)
Σ	3143.27	320.92(9.79)	167.95(18.7)	118.47(26.5)	112.60(27.9)	1771.94(1.77)

Table 8: Aurora execution times (in seconds) for the first set of benchmarks on TC2000.

The average real speedups on 32 processors, in comparison to SICStus on one TC2000 processor, are 17.6 for the *GI* programs, and 15.7 for the whole first set of benchmarks (calculated from Table 8). The corresponding figures for Muse, calculated from Table 3 (page 9), are 25.4 and 23.8 respectively. It can also be seen from Tables 2 (page 9), 3 (page 9) and 8 that Muse is faster than Aurora in all benchmarks. The Muse system has somewhat better speedups for hard benchmarks, i.e. benchmarks of groups *GII, GIII* and *GIV*. These hard benchmarks are a good test of the schedulers. The average real speedups on 32 processors of Aurora are 4.84 for *GII*, 0.81 for *GIII*, and 0.35 for *GIV*. The corresponding real speedups for Muse are 10.8, 2.2, and 0.73 respectively. These results illustrate that the Muse scheduler performs better by at least a factor 2.

Table 9 shows the ratio of the execution times on Aurora to the execution times on Muse for each group of the first set of benchmarks. Aurora timings are longer than Muse timings by 39% to 171% for 1 to 32 workers.

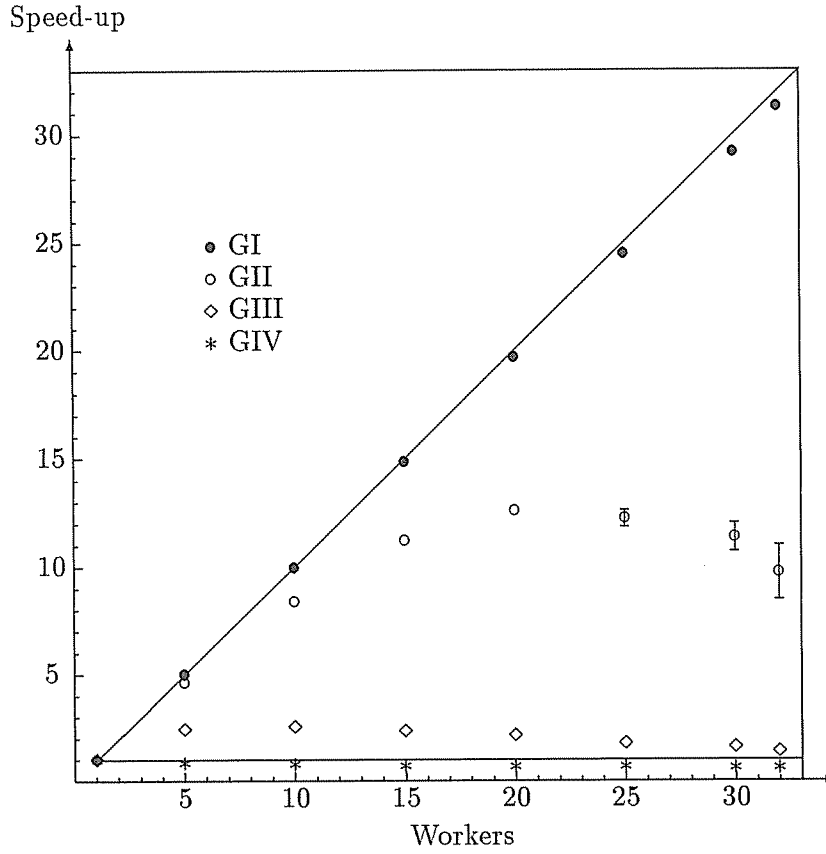

Figure 4: Speedups of Aurora for the first set of benchmarks on TC2000.

Benchmarks	Workers 1	10	20	30	32
GI	1.46	1.45	1.46	1.45	1.44
GII	1.52	1.48	1.54	1.89	2.23
GIII	1.38	1.57	1.81	2.38	2.71
GIV	1.38	1.68	1.95	2.01	2.08
Σ	1.46	1.46	1.48	1.50	1.51

Table 9: The Aurora to Muse ratio of execution times for the first set of benchmarks on TC2000.

Tables 10 and 11 present the execution times (in seconds) from the execution of the second set of benchmarks for Aurora and Muse on the TC2000 machine. Figures 5 and 6 show the corresponding speedup curves for the two groups of the second set of benchmarks: *mm* and *num*. The reason there is considerable variation around the mean values on the cut curves is that the two systems (Muse and Aurora) do not make any special treatment of speculative work.

Benchmarks	Aurora Workers					
	1	10	20	30	32	SICStus0.6
mm1	1.69	0.80(2.11)	0.68(2.49)	0.75(2.25)	1.39(1.22)	0.88(1.92)
mm2	1.23	0.63(1.95)	0.58(2.12)	0.43(2.86)	0.45(2.73)	0.69(1.78)
mm3	3.63	1.28(2.84)	1.03(3.52)	1.08(3.36)	1.16(3.13)	1.99(1.82)
mm4	6.39	1.59(4.02)	1.11(5.76)	1.22(5.24)	1.12(5.71)	3.36(1.90)
○ mm Σ	12.94	4.28(3.02)	3.36(3.85)	3.38(3.83)	3.92(3.30)	6.92(1.87)
num1	0.52	0.19(2.74)	0.14(3.71)	0.15(3.47)	0.16(3.25)	0.36(1.44)
num2	0.84	0.19(4.42)	0.13(6.46)	0.16(5.25)	0.18(4.67)	0.59(1.42)
num3	0.90	0.15(6.00)	0.11(8.18)	0.13(6.92)	0.13(6.92)	0.63(1.43)
num4	1.16	0.16(7.25)	0.12(9.67)	0.14(8.29)	0.17(6.82)	0.82(1.41)
◇ num Σ	3.42	0.68(5.03)	0.50(6.84)	0.57(6.00)	0.63(5.43)	2.40(1.43)
Σ	16.36	4.97(3.29)	3.86(4.24)	3.97(4.12)	4.56(3.59)	9.32(1.76)

Table 10: Aurora execution times (in seconds) for the second set of benchmarks on TC2000.

Benchmarks	Muse Workers					
	1	10	20	30	32	SICStus0.6
mm1	1.18	0.29(4.07)	0.25(4.72)	0.21(5.62)	0.22(5.36)	0.88(1.34)
mm2	0.92	0.21(4.38)	0.15(6.13)	0.14(6.57)	0.14(6.57)	0.69(1.33)
mm3	2.67	0.68(3.93)	0.50(5.34)	0.44(6.07)	0.45(5.93)	1.99(1.34)
mm4	4.57	0.72(6.35)	0.45(10.2)	0.40(11.4)	0.39(11.7)	3.36(1.36)
• mm Σ	9.34	1.89(4.94)	1.34(6.97)	1.18(7.92)	1.20(7.78)	6.92(1.35)
num1	0.45	0.11(4.09)	0.08(5.62)	0.08(5.62)	0.09(5.00)	0.36(1.25)
num2	0.74	0.14(5.29)	0.11(6.73)	0.11(6.73)	0.11(6.73)	0.59(1.25)
num3	0.79	0.11(7.18)	0.08(9.88)	0.09(8.78)	0.08(9.88)	0.63(1.25)
num4	1.02	0.12(8.50)	0.08(12.8)	0.09(11.3)	0.09(11.3)	0.82(1.24)
⋆ num Σ	3.00	0.48(6.25)	0.35(8.57)	0.36(8.33)	0.37(8.11)	2.40(1.25)
Σ	12.34	2.36(5.23)	1.69(7.30)	1.54(8.01)	1.57(7.86)	9.32(1.32)

Table 11: Muse execution times (in seconds) for the second set of benchmarks on TC2000.

It can be seen from Tables 10 and 11 that Muse is faster than Aurora for all benchmarks in the second set. The Muse system also has better speedups for all benchmarks. The last row of Table 12 shows the ratio of the execution times on Aurora

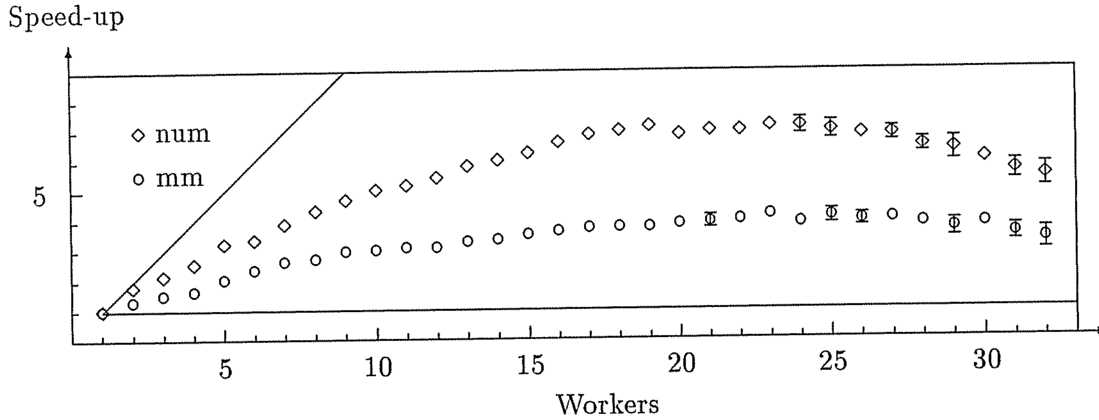

Figure 5: Speedups of Aurora for the second set of benchmarks on TC2000.

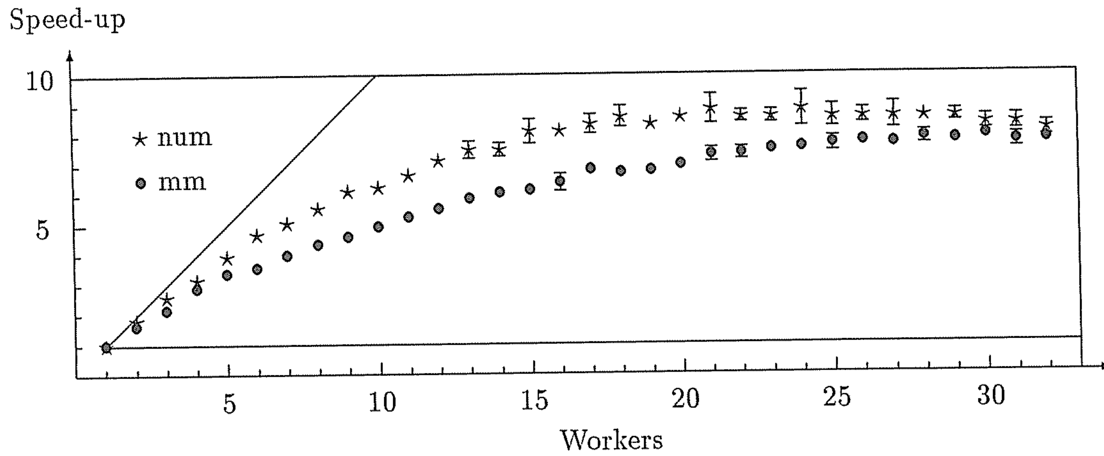

Figure 6: Speedups of Muse for the second set of benchmarks on TC2000.

to the execution times on Muse for the entire second set of benchmarks. Aurora timings are longer than Muse timings by 33% to 190% for 1 to 32 workers.

6 Conclusions

The Muse implementation and its performance results on the Butterfly machines have been presented and discussed. A large set of benchmarks with different amounts of parallelism have been used in the evaluation of Muse performance on the Butterfly machines. The performance results of Muse have also been compared with the corresponding results for the Aurora OR-parallel Prolog system.

The results obtained for the Muse system are very encouraging: almost linear

System	Workers 1	10	20	30	32
Aurora	16.36	4.97	3.86	3.97	4.56
Muse	12.34	2.36	1.69	1.54	1.57
Aurora/Muse	1.33	2.11	2.28	2.58	2.90

Table 12: Aurora and Muse execution times (in seconds) and the ratio between them for the second set of benchmarks on TC2000.

speedups (around 60 on 64 GP1000 processors, and 31 on 32 TC2000 processors) for Prolog programs with coarse grain parallelism, and a speedup factor of almost 1 for programs with very low parallelism. The average real speedup on 64 GP1000 processors, in a comparison with SICStus on one GP1000 processor, is 55.2 for the programs with coarse grain parallelism. The corresponding average real speedup on 32 TC2000 processors is 25.4. The obtained speedups, for the same number of processors, on the GP1000 are better than those obtained on the TC2000. This is due to the difference in the relative speed of the engine and the scheduler on the two machines. This relative speed is better on the TC2000, due to the fact that most engine data are cachable. This is not a factor on the GP1000 as it does not have cache memory.

The overhead of Muse associated with adapting the sequential SICStus Prolog system to an OR-parallel implementation is low, around 8% on the GP1000 and around 22% on the TC2000. The higher overhead on the TC2000 is due to the fact that each of its processors is provided with cache memory while the GP1000 processor is not. In Muse the Prolog tables are not cachable while in SICStus system all code, data, and tables are cachable. In Muse the Prolog tables are partitioned into sections and each section resides in the local memory of a processor. Access to cache memory is much faster than to local (and remote) memory.

The corresponding overhead per worker for the Aurora system on TC2000 is around 77%. One reason why the overhead per worker is higher in Aurora (than in Muse) on the TC2000 is that in Aurora the WAM stacks are shared by the all workers, and thus the stacks are not cachable. In Muse each worker has its own copy of the WAM stacks, and thus it is straightforward to make the WAM stacks cachable. There is also the overhead of maintaining the binding arrays used in Aurora. The average real speedup on 32 TC2000 processors, for the programs with coarse grain parallelism, is 17.6 for Aurora and 25.4 for Muse. For programs with finer parallelism, the real speedups are clearly in favor of the Muse system by at least a factor 2. For all benchmarks used in this paper, the Muse system is faster than the Aurora system.

The delayed release mechanism supported by the Muse system avoids a continuous decrease in task sizes as the number of workers grows. Without such a mechanism the gains due to parallel execution can be lost as the number of workers is increased,

due to the increasing overheads for scheduling work because of decreasing task sizes. The scheduling overhead of the Muse system on TC2000 for a set of benchmarks is around 8 – 26 Prolog procedure calls per task. The corresponding cost for Muse on Sequent Symmetry is around 5 – 7 Prolog procedure calls per task. The relative cost of scheduling overhead is higher on TC2000 than on Sequent Symmetry due to the relatively high ratio of processor speed to communication speed in the former. Thus, the task size should be controlled at runtime to be not less than 26 Prolog procedure calls on TC2000 in order to avoid performance degradation.

In the near future we plan to support handling of speculative work better on the Muse system by using one of the schemes presented in [14]. The Muse group at SICS is collaborating with a group at BIM to extend the BIM sequential Prolog system to an OR-parallel implementation using the Muse approach. We believe that extending the BIM Prolog system will be as simple and efficient as extending the SICStus Prolog system.

7 Acknowledgments

We would like to thank the Argonne National Laboratory group for allowing us to use their Butterfly machines. Shyam Mudambi was supported by NSF grant CCR-8718989.

References

[1] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445 – 475, December 1990.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[3] Khayri A. M. Ali and Roland Karlsson. The Muse OR-parallel Prolog Model and its Performance. In *the Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[4] Khayri A. M. Ali and Roland Karlsson. Scheduling OR-parallelism in Muse. In *the Proceedings of the 1991 International Conference on Logic Programming*, Paris, June 1991.

[5] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An

Overview and Evaluation Results. In *the Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 841 – 850. ICOT, November 1988.

[6] Prasenjit Biswas, Shyh-Chang Su, and David Y. Y. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND parallelism (RAP) in Logic Programs. In *the Proceedings of the Fifth International Conference' and Symposium on Logic Programming*, pages 1160–1179. MIT Press, August 1988.

[7] Ralph Butler, Ewing Lusk, Robert Olson, and Ross Overbeek. ANLWAM—A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, 1986.

[8] Alan Calderwood and Péter Szeredi. Scheduling OR-parallelism in Aurora—the Manchester Scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[9] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual (for version 0.6). SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.

[10] William Clocksin. Principles of the DelPhi Parallel Inference Machine. *Computer Journal*, 30(5):386–392, 1987.

[11] John Conery. Binding Environments for Parallel Logic Programs in Non-shared Memory Multiprocessors. *International Journal of Parallel Programming*, 17(2):125–152, April 1988.

[12] Doug DeGroot. Restricted AND-Parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[13] Bogumił Hausman. Handling of Speculative Work in OR-parallel Prolog: Evaluation Results. In *the Proceedings of the 1990 North American Conference on Logic Programming*, pages 721–736. MIT Press, October 1990.

[14] Bogumił Hausman. *Pruning and Speculative Work in OR-parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[15] Manuel Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 25–39. Springer-Verlag, 1986.

[16] Laxmikant V. Kalé. The Reduce-OR Process Model for Parallel Evaluation of Logic Programs. In *the Proceedings of the Fourth International Conference on Logic Programming*, pages 616–632. MIT Press, May 1987.

[17] Laxmikant V. Kalé, B. Ramkumar, and W. Shu. A Memory Organization Independent Binding Environment for And and Or Parallel Execution of Logic Programs. In *the Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1223–1240, 1988.

[18] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma. KABU-WAKE: A New Parallel Inference Method and Its Evaluation. *COMPCON Spring 86*, 1986.

[19] Yow-Jian Lin and Vipin Kumar. AND-parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *the Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141, 1988.

[20] Ewing Lusk and Robert McFadden. Using Automated Reasoning Tools: A Study of the Semigroup F2B2. *Semigroup Forum*, 36(1):75–88, 1987.

[21] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora OR-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[22] Shyam Mudambi. Performance of Aurora on NUMA machines. In *the Proceedings of the 1991 International Conference on Logic Programming*, pages 793–806. MIT Press, June 1991.

[23] Péter Szeredi. Performance Analysis of the Aurora OR-parallel Prolog System. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

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

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

[26] Harald Westphal, Philippe Robert, Jacques Chassin, and Jean-Claude Syre. The PEPSys model: combining backtracking, AND- and OR-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*. IEEE, 1987.

[27] H. Yasuhara and K. Nitadori. ORBIT: A Parallel Computing Model of Prolog. *New Generation Computing*, 2(3):277–288, 1984.