Experiments with ROO, a Parallel Automated Deduction System*

Ewing L. Lusk
William W. McCune

Mathematics and Computer Science Division
Argonne National Laboratory

lusk@mcs.anl.gov
mccune@mcs.anl.gov

1 Introduction

The automated theorem prover OTTER[11, 13] represents the state of the art in high-speed, general purpose theorem provers. One way to increase OTTER's speed further is through the exploitation of parallelism. A general, parallel algorithm for the computing the closure of a set under an operation was presented in [17]. Since OTTER's fundamental algorithm can be viewed as a closure computation, this algorithm can be applied to OTTER. The result is ROO, a parallel theorem prover compatible with OTTER that runs on shared-memory multiprocessors.

ROO itself is described in [7] and [8]. For completeness we present a summary of the basic algorithm in Section 2. Compared with the numerical applications typically run on today's multiprocessors, ROO's behavior is considerably more complex. Its algorithm performs well in general, but in certain situations does badly. Some of these situations reflect only certain phases of runs on particular problems. Sometimes ROO, even with only one process, outperforms OTTER, and sometimes it does much worse. Speedups are often roughly linear, which is why we are well-satisfied with ROO. Sometimes they are far below linear, and sometimes startlingly superlinear. In general, a full appreciation of the subtleties of parallel computation in this application area can only be obtained by looking closely at the behavior of ROO on a wide variety of theorem-proving problems.

The purpose of this paper is to provide such a detailed look. After summarizing the algorithms of OTTER and ROO in Section 2, we present a series of experiments taken from a wide variety of test problems. These show ROO both at its best and its worst and exhibit a number of surprising features. Since the problems themselves are of interest, we provide (except in one rather tedious case) the complete set of input clauses that make up the problem. For each problem we exhibit the performance of ROO and analyze the results.

Finally, the non-deterministic nature of parallel algorithms means that consecutive runs of the same input file, on the same number of processes, can produce different results. In Section 4 we address the question of the stability and reproducibility of the results we have reported for ROO.

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

2 Algorithms

2.1 A Sequential Theorem Proving Algorithm without Deletion

We present here a simplified form of the theorem-proving algorithm that has been used in Argonne systems over the years[23, 10, 11]. For an introduction to the approach, see [21]. We assume that the goal is to prove a set of clauses unsatisfiable. We assume further that the input clauses are divided into two sets, which we call *Usable* and *Set of support* (SOS), in such a way that the Usable set is satisfiable. (If the input clauses make this difficult to do, then the Usable set may be chosen to be the empty set.) Thus we will be using the "set-of-support strategy" first described in [22].

```
While (the null clause has not been produced and SOS is not empty)
    Choose a clause from the set of support, call it the given clause, and move it to
        the Usable set
    Generate all clauses that can be deduced from the given clause and
        other clauses in the Usable set
    For each new generated clause
        Process it (rewrite to canonical form, merge literals, etc.)
        Test if it is subsumed by any existing clause (in either Usable or SOS)
        Test if it is too heavy
        If the new clause survives
            Add it to SOS
        end if
    end for
end while
```

Figure 1: Sequential Theorem-Proving Algorithm without Deletion (Algorithm 1)

The algorithm given in Figure 1 can complete with the empty clause found (giving a proof that the input clause set is unsatisfiable), compete without the empty clause being found (no proof), or it can fail to terminate with the constraints of time and memory. If the filtering mechanism applied is appropriately chosen, then this scheme is complete: that is, if the algorithm terminates without finding the empty clause, then the original set of clauses is satisfiable.

The currently most effective implementation of this algorithm is OTTER[11]. OTTER has a wide variety of inference rules and control parameters for adapting this algorithm to a particular problem and sophisticated indexing methods to make each of the operations quite fast, even when the set of kept clauses has grown to a hundred thousand clauses or more.

This particular approach contrasts with some more recent "Prolog technology" theorem provers[18, 16], which use the compilation techniques from WAM-based Prolog implementations to achieve extremely high inference rates, at a cost of possibly redundant computations. The "closure" approach taken here is based on the expectation that a clause deduced and processed is worth keeping as a filter for preventing the deduction and subsequent use of duplicate or weaker clauses. Experience is on the side of the closure approach; although certain small problems can be done very quickly with the Prolog-technology approach, many

large problems done years ago with the closure approach remain out of reach of even the best Prolog technology systems. An example is given in Section 3.1 below.

2.2 ROO Without Deletion

In this section we present the parallel version of Algorithm 1 and its implementation based on OTTER.

An early attempt to parallelize Algorithm 1 focused on the inner "for" loop. This is relatively straightforward, since the rewriting, subsumption, and filtering of one new clause is independent from that of another, as long as one finishes up the loop by checking for subsumption among members of the batch. The problem with this approach is that even when there are large numbers of clauses in each batch, the barrier at the end of the loop meant that many processes were temporarily idle.

The key idea here is to parallelize the outer "while" loop instead. That is, we will consider multiple "given" clauses simultaneously. This both increases the grain size of the parallel computation and removes any barriers. The difficulty, of course, is that without some care, two copies of the same clause might enter the permanent clause space, each deduced and post-processed by a different process. We will also need to solve technical problems associated with adding clauses to the clause space while it is in use by other clauses.

Our approach is to use an intermediate holding area, which we will call (arbitrarily) K. New clauses are first put in K, from which they are removed by a single process which repeats the post-processing of the clause before adding it to the clause database. Thus we break down the work to be done into two tasks: A and B. At any moment multiple processes will be executing Task A, but at most one process will be executing Task B. Each instance of Task A is associated with a given clause. Task B is only executed when the set K is non-empty.

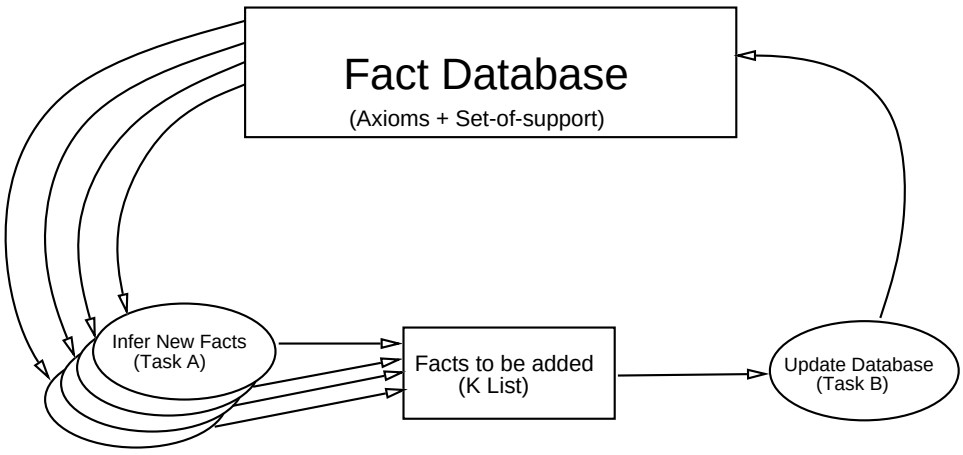

Figure 2: Flow of Data in Simplified ROO

Note that there is not a separate process dedicated to Task B. Rather, we create a uniform pool of processes, each of which performs the loop shown in Figure 5.

The loop continues until some process detects unit conflict or until the set of support is empty and all processes are waiting for a clause to appear there.

```
Task A(given clause):
    Generate new clauses
    For each new clause
        Rewrite it
        Subsumption test
        Filter
        If it survives
            Lock K
            Put new clause in K
            Unlock K
        end if
    end for
```

Figure 3: Task A

```
Task B:
    While K is not empty
        Lock K
        Choose a clause from K
        Unlock K
        Redo rewrite, subsumption test, filter
        If it survives
            Integrate new clause into database
            Put new clause in SOS
        end if
    end while
```

Figure 4: Task B

```
    While it is not time to stop
        If K is non-empty and no process is already doing it
            Do Task B
        else
            If SOS is not empty
                Choose a new given clause from SOS
                Do Task A(given clause)
            end if
        end if
    end while
```

Figure 5: Main loop performed by all processes

2.3 The Complete ROO Algorithm

In Section 2.2 we described a simplified version of OTTER's algorithm, in which no clauses are deleted from the database once they have been added. In the complete version, back subsumption tests are done on newly-derived clauses, causing deletions, and new rewrite rules may be derived in the course of the run. These new rewrite rules are immediately applied to existing clauses in the database, causing both deletions and new additions. Coping with deletions complexifies ROO since we do not want to interfere with the generation of new clauses when deleting. It turns out that this problem can be solved by having Task B handle actual deletions. Back subsumption and back demodulation processes can run in parallel with all other processes, but instead of actually deleting clauses from the database, they place their identifiers in shared lists where Task B (executed by only one process at a time) can find them and carry out the actual deletions. This algorithm is shown schematically in Figure 6.

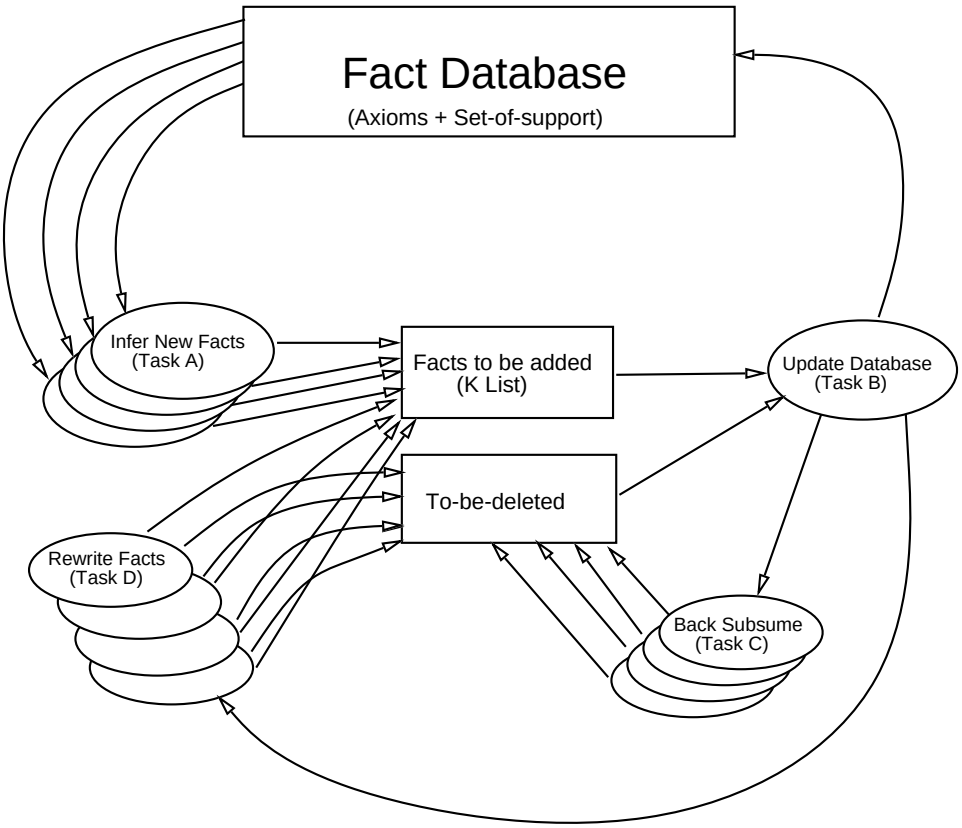

Figure 6: Flow of Data in ROO

The clause lists are held in shared memory with Otter's elaborate data structures for indexing. This makes the above algorithm very much a shared-memory parallel algorithm.

2.4 Some Important Features of OTTER and ROO

In order to understand the experiments, it is necessary to be familiar with at least a few of the many OTTER and ROO features for controlling the algorithms given in this section.

1. The user must select one or more inference rules. Each of the experiments presented here uses one of the inference rules hyperresolution, binary resolution, or paramodulation.

2. The default method for selecting the next given clause is by weight—to take one with the fewest number of symbols. An alternative is to use the SOS list as a queue, which results in a breadth-first search. The ratio strategy combines those two methods: The user specifies a ratio n, with n given clauses selected by weight for each selected because it is first in SOS.

3. The Knuth-Bendix option causes OTTER or ROO to use its paramodulation and demodulation as in the Knuth-Bendix completion method. With that option, the program can find refutations as well as terminate with a complete set of reductions. The user typically uses the lexicographic recursive path ordering (LRPO) and assigns an ordering on function symbols.

4. The rule for distinguishing variables from constants in clauses is that a symbol is a variable if and only if it starts with u–z.

2.5 Differences that Surface in the Experiments

The parallelism of ROO can significantly alter search spaces when compared to analogous OTTER runs. Both programs by default select the simplest clause in SOS as the given clause. However, at the start of parallel ROO searches, many instances of Task A demand given clauses before there are many to select from. The typical effect is that the first few kept clauses become given clauses, regardless of their complexities. (After the set of support has had a chance to grow, Task A can be more selective, and the behavior is similar to OTTER's.) The typical result of ROO's eagerness is that it takes a bit longer to find a proof; but in some cases those early, complex given clauses can lead to quick and short proofs, which are reflected in speedups that are superlinear, and in other cases, the complex clauses can lead to prolific but useless paths in the search space. To give a more accurate picture of the ROO's performance, we list speedups with respect to the number of clauses generated per second as well as for the run time.

Aside from the parallelism, there is another important difference between the ways that OTTER and ROO process generated clauses. OTTER processes generated clauses in the order in which they are generated. If a generated clause passes all of the retention tests, including forward subsumption, it is immediately integrated into the database and becomes available to subsume the next generated clause. In addition, if it is to become a demodulator, it does so immediately and is able to rewrite the next generated clause. When ROO's Task A generates a clause which passes the retention tests, the clause goes into limbo in the K list, and is not available to subsume or rewrite the next generated clause. ROO's Task B removes clauses from the K list shortest-first and integrates those that pass the second application of the retention tests. OTTER can perform better in some cases, because ROO must reapply retention tests, and ROO can perform better in other cases, because it integrates simple

clauses before complex clauses. This behavior occurs in one-processor ROO searches as well as in parallel searches.

3 Experiments

The following experiments come from a variety of areas: lattice theory, semigroup theory, group theory, Boolean algebra, circuit design, calculus, non-classical logic, and robotics. They include several standard theorem prover test problems from the literature as well as some more exotic ones.

The tests were conducted on the 26-processor, 32 megabyte Sequent Symmetry in the Advanced Computing Research Facility at Argonne National laboratory. We give results for OTTER as well as for ROO, (1) so that speedups can be measured against the best known sequential algorithm, as they should be, (2) to show that the parallel algorithm with one process is in general comparable to that of OTTER, and (3) to illustrate the occasional differences between OTTER and ROO with one process.

We give statistics not only for elapsed time (in seconds) but also for the number of clauses generated, because it is a rough measure of how much work was done during the run. We aim for regular and predictable speedups in the amount of work done per unit of time, even when the rearrangement of the search space caused by nondeterminism causes sudden changes in the time ROO takes to discover a proof.

3.1 SAM's Lemma

SAM's Lemma, a problem in lattice theory named after the first theorem prover that proved it, is a standard problem for evaluating theorem provers. It was one of the first non-trivial problems done by an automated system. It was first described in [3]. Although its solution was reported more than twenty years ago, it still remains beyond the reach of most current systems. The main reason is common to many problems in algebra: there are many paths to each derived clause, so any system that does not employ subsumption will explore a much larger search space than is necessary. (Clauses 1–20 below are not a full axiomatization of modular lattice theory.)

Clauses for SAM's Lemma

Input Usable Clauses

1 $join(1, x, 1)$.
2 $join(x, 1, 1)$.
3 $join(0, x, x)$.
4 $join(x, 0, x)$.
5 $meet(0, x, 0)$.
6 $meet(x, 0, 0)$.
7 $meet(1, x, x)$.
8 $meet(x, 1, x)$.
9 $meet(x, x, x)$.
10 $join(x, x, x)$.
11 $\neg meet(x, y, z) \mid meet(y, x, z)$.
12 $\neg join(x, y, z) \mid join(y, x, z)$.

13 $\neg meet(x,y,z) \mid join(x,z,x)$.
14 $\neg join(x,y,z) \mid meet(x,z,x)$.
15 $\neg meet(x,y,xy) \mid \neg meet(y,z,yz) \mid \neg meet(x,yz,xyz) \mid meet(xy,z,xyz)$.
16 $\neg meet(x,y,xy) \mid \neg meet(y,z,yz) \mid \neg meet(xy,z,xyz) \mid meet(x,yz,xyz)$.
17 $\neg join(x,y,xy) \mid \neg join(y,z,yz) \mid \neg join(x,yz,xyz) \mid join(xy,z,xyz)$.
18 $\neg join(x,y,xy) \mid \neg join(y,z,yz) \mid \neg join(xy,z,xyz) \mid join(x,yz,xyz)$.
19 $\neg meet(x,z,x) \mid \neg join(x,y,x1) \mid \neg meet(y,z,y1) \mid \neg meet(z,x1,z1) \mid join(x,y1,z1)$.
20 $\neg meet(x,z,x) \mid \neg join(x,y,x1) \mid \neg meet(y,z,y1) \mid \neg join(x,y1,z1) \mid meet(z,x1,z1)$.

Input Set of Support Clauses

21 $meet(a,b,c)$.
22 $join(c,r2,1)$.
23 $meet(c,r2,0)$.
24 $meet(r2,b,e)$.
25 $join(a,b,c2)$.
26 $join(c2,r1,1)$.
27 $meet(c2,r1,0)$.
28 $meet(r2,a,d)$.
29 $join(r1,e,a2)$.
30 $join(r1,d,b2)$.
31 $\neg meet(a2,b2,r1)$.

Strategy for SAM's Lemma

The inference rule was hyperresolution, and forward subsumption (but not back subsumption) was applied.

Results for SAM's Lemma

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	30.70	32.04	7.91	4.37	2.57	2.85	1.98	1.62
Generated	5924	5981	5977	6079	6022	6208	6143	5837
Kept	134	134	131	131	130	130	129	124
Memory (K)	95	159	220	344	468	592	716	840
Gen/sec	192	186	755	1391	2343	2178	3102	3603
Speedups:								
Proof	1.0	1.0	3.9	7.0	11.9	10.8	15.5	19.0
Gen/sec	1.0	1.0	3.9	7.2	12.2	11.3	16.2	18.8

Notes on SAM's Lemma

The performance of Roo is excellent with up to 12 processes, because of the simple and uniform nature of the clauses. Speedups with more than 12 processes are less than linear because the problem is so short.

3.2 Imp-4

This problem in the implicational propositional calculus was first brought to our attention in [14]. Clause 2 below is a single axiom (due to Łukasiewicz) for the calculus, clause 3

denies the law of hypothetical syllogism, and clause 1 is the representation of condensed detachment. Any refutation of these three clauses shows that law of hypothetical syllogism can be derived by condensed detachment from the single axiom.

Clauses for Imp-4

Input Usable Clauses

1 $\neg P(x) \mid \neg P(i(x,y)) \mid P(y)$.

Input Set of Support Clauses

2 $P(i(i(i(x,y),z),i(i(z,x),i(u,x))))$.
3 $\neg P(i(i(a,b),i(i(b,c),i(a,c))))$.

Strategy for Imp-4

The inference rule was hyperresolution, and forward subsumption (but not back subsumption) was applied. In addition, to conserve memory, generated clauses with more than 20 symbols were discarded.

Results for Imp-4

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	29098.32	29984.67	7180.29	3440.89	2462.52	1844.03	1492.35	1269.28
Generated	6706380	6668046	6413924	6208570	6649996	6662397	6826296	7108289
Kept	20410	20342	20309	20242	20438	18281	19534	18759
Memory (K)	7185	7791	8653	9737	13119	14235	17109	19498
Gen/sec	230	222	893	1804	2700	3612	4574	5600
Speedups:								
Proof	1.0	1.0	4.1	8.5	11.8	15.8	19.5	22.9
Gen/sec	1.0	1.0	3.9	7.8	11.7	15.7	19.9	24.3

Notes on Imp-4

ROO obtains nearly linear speedups on Imp-4. Note that with 24 processes, more clauses are generated, but fewer are kept than in most other cases.

3.3 F3B2

It is possible for a free semigroup to be known to be finite yet for its size to remain unknown. The semigroup can be represented in terms of generators and an operation, in which the semigroup itself is the closure of the set of generators under that operation. OTTER's (and therefore ROO's) basic algorithm can thus be used to compute the size of the semigroup. It is only necessary to describe the semigroup operation in terms of hyperresolution. this process is described in detail in [9] for the case of the semigroup F2B2. The semigroup F3B2 is larger, and potentially contains 5^{125} elements. The fact that there are actually only 1435 elements was first discovered by ITP, OTTER's predecessor[6, 5]. Again, this problem relies heavily on subsumption.

Clauses for F3B2

The elements of F3B2 are 125-tuples of elements of the Bradt semigroup B2, whose multiplication table is as follows:

	0	e	f	a	b
0	0	0	0	0	0
e	0	e	0	a	0
f	0	0	f	0	b
a	0	0	a	0	e
b	0	b	0	f	0

Multiplication in F3B2 is componentwise. The generators of F3B2 can be written as

$g_1 = 0^5 e^5 f^5 a^5 b^5$
$g_2 = (0efab)^5$
$h_1 = 0^5 e^5 f^5 b^5 a^5$
$h_2 = (0efba)^5$

To find the size of F3B2, we use clauses (1) that say that these are the generators, (2) that say that generators are elements of the semigroup, and (3) that say that the product of a generator and an element is an element, where the product is defined by demodulators. The complete set of clauses is a straightforward extension of the clauses given in [9] for F2B2.

Strategy for F3B2

The inference rule was hyperresolution, and forward subsumption (but not back subsumption) was applied. No refutation is expected—the run terminates when the SOS list is exhausted.

Results for F3B2

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20
Run time	1056.74	1062.74	277.06	142.83	98.62	77.80	65.97
Generated	8616	8616	8616	8616	8616	8616	8616
Kept	1435	1435	1435	1435	1435	1435	1435
Memory (K)	6642	6706	8716	11270	14652	18096	23139
Gen/sec	8	8	31	60	87	110	130
Speedups:							
Run time	1.0	1.0	3.8	7.4	10.7	13.6	16.0
Gen/sec	1.0	1.0	3.9	7.5	10.9	13.8	16.3

Notes on F3B2

Performance of Roo is very regular, but we have not yet determined why the speedups are not closer to being linear.

3.4 2-Inverter Problem

Is it possible to construct a 3-input, 3-output binary circuit, consisting of any number of AND gates and OR gates, but only two NOT gates, in which each output is the inversion

of the corresponding input? This is a pleasing circuit design puzzle, described in [21] and in [15]. The trick is to keep track not of every circuit that can be built, but only of the output patterns that can be constructed, and the use of the inverters. An interesting result derived by OTTER is that any solution of this problem (of which there are many) must use the two inverters in the same way; that is, to invert the same signals.

Clauses for 2-Inverter Problem

Input Usable Clauses

1 $\neg function(x,v) \mid \neg function(y,v) \mid function(\$BIT\_AND(x,y),v)$.
2 $\neg function(x,v) \mid \neg function(y,v) \mid function(\$BIT\_OR(x,y),v)$.
3 $\neg function(x,v) \mid function(\$BIT\_AND(255,\$BIT\_NOT(x)),$
$addinv(v,invtab(\$BIT\_AND(255,\$BIT\_NOT(x)))))$.
4 $\neg function(240,v) \mid \neg function(204,v) \mid \neg function(170,v)$.

Input Set of Support Clauses

5 $function(15,v)$.
6 $function(51,v)$.
7 $function(85,v)$.

Input Demodulators

8 $(addinv(l(x,y),z) = l(x,addinv(y,z)))$.
9 $\$CONDITIONAL(\$VAR(x),addinv(x,y),l(y,x))$.

Strategy for 2-Inverter Problem

The clauses for this problem use several built-in functions (those that start with $\$BIT$) to perform bitwise operations. Boolean functions are encoded as integers, and demodulation evaluates the built-in functions. Demodulators 8 and 9 manage the list of inverted signals. $\$CONDITIONAL(\alpha,\beta,\gamma)$ means "if α, then rewrite β to γ".

The inference rule was hyperresolution, and forward subsumption (but not back subsumption) was applied. Generated clauses representing circuits with more than two inverters were discarded by weighting.

Results for 2-Inverter Problem

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	47235.57	47704.68	12376.65	6219.20	4402.19	3141.14	2586.23	2236.77
Generated	6323644	6324501	6313683	6050605	6340536	6099702	5978269	6351410
Kept	21342	21343	21343	21343	21344	21343	21343	21343
Memory (K)	5588	5875	6033	6256	6762	6959	16184	18736
Gen/sec	133	132	510	972	1440	1941	2311	2839
Speedups:								
Proof	1.0	1.0	3.8	7.6	10.7	15.0	18.3	21.1
Gen/sec	1.0	1.0	3.8	7.3	10.8	14.6	17.4	21.3

Notes on 2-Inverter Problem

Performance of ROO is regular, with excellent speedups.

3.5 Bledsoe-P1

This is first (easiest) of a sequence of five challenge problems from W. Bledsoe[1]. All of the five problems are variants of the theorem that the sum of continuous functions is continuous.

Clauses for Bledsoe-P1

Input Usable Clauses

1 $\neg LE(0, xE) \mid LE(0, D1(xE))$.
2 $\neg LE(0, xE) \mid LE(0, D2(xE))$.
3 $\neg LE(0, xE) \mid \neg LE(abs(+(x, \neg(A))), D1(xE)) \mid LE(abs(+(F(x), \neg(F(A)))), xE)$.
4 $\neg LE(0, xE) \mid \neg LE(abs(+(x, \neg(A))), D2(xE)) \mid LE(abs(+(G(x), \neg(G(A)))), xE)$.
5 $LE(0, E0)$.
6 $\neg LE(0, xD) \mid LE(abs(+(XS(xD), \neg(A))), xD)$.
7 $\neg LE(z, MIN(x, y)) \mid LE(z, x)$.
8 $\neg LE(z, MIN(x, y)) \mid LE(z, y)$.
9 $\neg LE(0, x) \mid \neg LE(0, y) \mid LE(0, MIN(x, y))$.
10 $\neg LE(x, HALF(z)) \mid \neg LE(y, HALF(z)) \mid LE(+(x, y), z)$.
11 $\neg LE(0, x) \mid LE(0, HALF(x))$.

Input Set of Support Clauses

12 $\neg LE(0, xD) \mid$
$\neg LE(+(abs(+(F(XS(xD)), \neg(F(A)))), abs(+(G(XS(xD)), \neg(G(A))))), E0)$.

Strategy for Bledsoe-P1

The inference rule was binary resolution with factoring. (Factoring is not required). Forward subsumption (but not back subsumption) was applied. Generated clauses with more than 40 symbols were discarded, and generated clauses in which any of the function symbols *HALF*, *abs*, *MIN*, +, $D1$, or $D2$ was nested with itself were discarded.

Results for Bledsoe-P1

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	49.63	66.67	18.04	11.37	11.41	11.19	11.89	12.37
Generated	2594	2338	3056	3235	3136	3156	3143	3122
Kept	756	704	652	523	523	524	524	522
Memory (K)	510	574	1372	1880	1940	2000	2028	2216
Gen/sec	52	35	169	284	274	282	264	252
Speedups from Otter:								
Proof	1.0	0.7	2.8	4.4	4.3	4.4	4.2	4.0
Gen/sec	1.0	0.7	3.3	5.5	5.3	5.4	5.1	4.9
Speedups from Roo-1:								
Proof	1.3	1.0	3.7	5.9	5.8	6.0	5.6	5.4
Gen/sec	1.5	1.0	4.8	8.1	7.9	8.1	7.6	7.2

Notes on Bledsoe-P1

The difference in performance between OTTER and ROO-1 results from extra forward subsumption tests done by ROO-1. Generated clauses were typically nonunit, and subsumption with nonunit clauses is very expensive. Statistics in the output files showed that Task B was a bottleneck, requiring more than 10 seconds in each of the ROO runs, which is reflected the poor speedups.

3.6 CD-12

Problem CD-12 is from the two-valued sentential calculus. Clauses 2–4 below are Łukasiewicz's axiomatization of the calculus, clause 5 denies a form of the law of syllogism, and clause 1 encodes condensed detachment. The symbols i and n can be interpreted as implication and negation, respectively.

Clauses for CD-12

Input Usable Clauses

1 $\neg P(i(x,y)) \mid \neg P(x) \mid P(y).$

Input Set of Support Clauses

2 $P(i(i(x,y),i(i(y,z),i(x,z)))).$
3 $P(i(i(n(x),x),x)).$
4 $P(i(x,i(n(x),y))).$
5 $\neg P(i(i(b,c),i(i(a,b),i(a,c)))).$

Strategy for CD-12

The inference rule was hyperresolution, and both forward and back subsumption were applied. Generated clauses with more than 20 symbols were discarded.

Results for CD-12

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	373.96	398.19	108.89	54.30	27.13	24.82	8.42	8.15
Generated	78301	78303	79323	84353	67595	80796	31972	30692
Kept	7220	7122	7451	5183	2951	2480	803	612
Memory (K)	3576	3768	5459	7309	6668	8998	6188	6312
Gen/sec	209	196	728	1553	2491	3255	3797	3765
Speedups:								
Proof	1.0	0.9	3.4	6.9	13.8	15.1	44.4	45.9
Gen/sec	1.0	0.9	3.5	7.4	11.9	15.6	18.2	18.0

Notes on CD-12

Performance of ROO appears to be regular with up to 16 processes. ROO-20 and ROO-24 apparently found shortcuts in the search spaces. (See Sections 2.5 and 4).

3.7 CD-13

Problem CD-13 is similar to CD-12, except that the goal is to prove Peirce's law (denied in clause 5 below) rather than the syllogism law.

Clauses for CD-13

Input Usable Clauses

1 $\neg P(i(x,y)) \mid \neg P(x) \mid P(y)$.

Input Set of Support Clauses

2 $P(i(i(x,y),i(i(y,z),i(x,z))))$.
3 $P(i(i(n(x),x),x))$.
4 $P(i(x,i(n(x),y)))$.
5 $\neg P(i(i(i(a,b),a),a))$.

Strategy for CD-13

The inference rule was hyperresolution, and both forward and subsumption were applied. Generated clauses with more than 20 symbols were discarded.

Results for CD-13

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	373.07	416.12	104.20	58.68	34.21	9.38	7.62	18.78
Generated	78624	78773	80606	88026	83701	29850	29385	90541
Kept	7324	7292	7149	5270	3546	1016	739	1533
Memory (K)	3608	3800	5171	8138	8649	5615	5836	12220
Gen/sec	210	189	773	1500	2446	3182	3856	4821
Speedups:								
Proof	1.0	0.9	3.6	6.4	10.9	39.8	49.0	19.9
Gen/sec	1.0	0.9	3.7	7.1	11.7	15.2	18.4	23.0

Notes on CD-13

Performance of ROO appears to be regular with up to 12 processes, but ROO-16 and ROO-20 apparently found shortcuts in the search spaces, and ROO-24 had to explore more of the space. (See Sections 2.5 and 4).

3.8 CD-90

Problem CD-90, which is in the left group calculus[4], is to show a dependence in one of J. Kalman's original axiomatizations of the calculus[12]. The symbols P and e can be interpreted as "is the identity" and left division in groups.

Clauses for CD-90

Input Usable Clauses

1 $\neg P(e(x,y)) \mid \neg P(x) \mid P(y)$.

Input Set of Support Clauses

2 $P(e(e(e(e(x,y),e(x,z)),e(y,z)),u),u))$.
3 $P(e(e(e(e(e(x,y),e(x,z)),u),e(e(y,z),u)),v),v))$.
4 $\neg P(e(e(e(e(a,b),c),d),e(e(e(a,p),c),e(e(b,p),d))))$.

Strategy for CD-90

The inference rule was hyperresolution, and both forward and subsumption were applied. Generated clauses with more than 20 symbols were discarded.

Results for CD-90

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20
Run time	825.46	867.62	221.31	120.80	156.27	146.37	160.18
Generated	149798	149806	151714	153693	261292	380548	542331
Kept	5909	5909	5951	5981	6047	6091	6100
Memory (K)	3097	3257	3766	6191	10309	14396	16821
Gen/sec	181	172	685	1272	1672	2599	3385
Speedups:							
Proof	1.0	1.0	3.7	6.8	5.3	5.6	5.2
Gen/sec	1.0	1.0	3.8	7.0	9.2	14.4	18.7

Notes on CD-90

Performance of Roo appears to be regular with up to 8 processes, but with more processes, Roo's eagerness appears to have led it down fruitless paths.

3.9 Apabhp

This is the "blind hand problem", a standard problem from the area of robotics. It is included in many theorem prover test suites.

Clauses for Apabhp

Input Set of Support Clauses

1 $\neg a(m(x),z,d(g(z),y)) \mid a(m(x),z,y) \mid i(m(x),y)$.
2 $\neg a(m(x),z,y) \mid \neg a(h,z,y) \mid i(m(k(y)),d(p,y))$.
3 $\neg a(h,z,y) \mid a(m(x),z,y) \mid \neg i(m(x),y)$.
4 $\neg a(m(x),z,y) \mid a(h,z,y) \mid \neg i(m(x),y)$.
5 $\neg a(x,t,y) \mid c(y) \mid \neg r(x)$.
6 $\neg a(m(x),z,y) \mid a(m(x),z,d(g(z),y))$.
7 $\neg i(m(x),y) \mid i(m(x),d(g(z),y))$.
8 $\neg a(h,z,y) \mid a(m(k(y)),z,y)$.
9 $\neg a(x,z,y) \mid a(x,z,d(p,y))$.
10 $\neg a(x,z,y) \mid a(x,z,d(l,y))$.
11 $\neg a(x,z,d(l,y)) \mid a(x,z,y)$.
12 $\neg a(x,e,n) \mid r(x)$.
13 $\neg c(y)$.

14 $\neg a(x,e,y) \mid \neg a(x,t,y)$.
15 $\neg i(m(x),d(l,y))$.
16 $\neg r(h)$.
17 $a(h,z,d(g(z),y))$.
18 $a(m(s),e,n)$.

Strategy for Apabhp

The inference rule was hyperresolution, and both forward and subsumption were applied. Given clauses were selected with ratio 3 (Section 2.4).

Results for Apabhp

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	55.55	46.94	13.10	5.48	5.22	5.72	5.00	5.37
Generated	3103	2312	2485	2639	2583	2554	2564	2565
Kept	1621	1285	1403	583	495	493	494	494
Memory (K)	894	798	2329	2584	2580	2576	2732	2856
Gen/sec	55	49	189	481	494	446	512	477
Speedups:								
Proof	1.0	1.2	4.2	10.1	10.6	9.7	11.1	10.3
Gen/sec	1.0	0.9	3.4	8.8	9.0	8.1	9.3	8.7

Notes on Apabhp

The reason Roo-1 performs better than Otter is that generated clauses from a given clause are integrated smallest-first in Roo, and in this problem, given clauses generate many clauses that properly subsume one another. The integration of smaller (more general) clauses first prevents the integration of larger clauses they subsume.

Speedups are limited by Task B, which requires at least 4.2 seconds in each of the runs.

3.10 Robbins

A proof for this problem shows that if the hypothesis $\exists x, x + x = x$ is added to a Robbins algebra, then the resulting algebra is Boolean[20]. Clauses 2–4 axiomatize Robbins algebra, clause 5 asserts the hypotheses, and clause 6 denies Huntington's axiom. (Huntington's axiom together with clauses 2 and 3 axiomatize Boolean algebra.)

Clauses for Robbins

Input Usable Clauses

1 $(x = x)$.
2 $(+(x,y) = +(y,x))$.
3 $(+(+(x,y),z) = +(x,+(y,z)))$.

Input Set of Support Clauses

4 $(n(+(n(+(x,y)),n(+(x,n(y))))) = x)$.
5 $(+(C,C) = C)$.
6 $(+(n(+(A,n(B))),n(+(n(A),n(B)))) \neq B)$.

Strategy for Robbins

The Knuth-Bendix option was used, with LRPO ($+ > n > C > B > A$, and $+$ with LR status). Generated and simplified clauses with more than 20 symbols were discarded. Two sets of experiments were run: (1) selecting given clauses by symbol count, and (2) selecting given clauses with ratio 3 (Section 2.4).

Results for Robbins (without Ratio)

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	349.17	320.41	72.94	10.60	9.93	11.23	10.89	14.82
Generated	8272	8874	11430	2961	4725	7567	10271	17366
Kept	456	401	451	257	263	274	236	385
Memory (K)	638	734	1116	1016	1396	1776	2284	2952
Gen/sec	23	27	156	279	475	673	943	1171
Speedups:								
Proof	1.0	1.1	4.8	32.9	35.2	31.1	32.1	23.6
Gen/sec	1.0	1.2	6.8	12.1	20.7	29.3	41.0	50.9

Results for Robbins (with Ratio 3)

	OTTER	ROO-1	ROO-4	ROO-8	ROO-12	ROO-16	ROO-20	ROO-24
Run time	104.57	156.73	28.77	23.88	26.20	19.53	21.69	15.73
Generated	3543	4867	4302	7354	13717	13665	18793	17698
Kept	315	351	301	390	428	425	436	363
Memory (K)	479	670	892	1336	1972	2192	2508	2664
Gen/sec	33	31	149	307	523	699	866	1125
Speedups:								
Proof	1.0	0.7	3.6	4.4	4.0	5.4	4.8	6.6
Gen/sec	1.0	0.9	4.5	9.3	15.9	21.2	26.3	34.1

Notes on Robbins

Note that the poor speedups in the ratio 3 experiments when compared to the experiments without ratio are due mostly to the improvement in OTTER performance.

3.11 Z22

This is a problem in combinatorial group theory first brought to our attention by J. Kalman. It is one of a family of problems to determine the structure of certain groups, called Fibonacci groups, defined by a symmetric set of relations. Some of these problems are still open. It illustrates the performance of ROO on a Knuth-Bendix completion problem. The problem is to show that the group on five generators with the five given relations (clauses 12–16) is the cyclic group of order 22. We use a notation in which multiplication on the left is expressed as the application of a function, and $a1, b1 \ldots$ represent the inverses of $a, b, \ldots$.

Clauses for Z22

Input Usable Clauses

1 $(x = x)$.

Input Set of Support Clauses

2 $(a(a1(x)) = x)$.
3 $(a1(a(x)) = x)$.
4 $(b(b1(x)) = x)$.
5 $(b1(b(x)) = x)$.
6 $(c(c1(x)) = x)$.
7 $(c1(c(x)) = x)$.
8 $(d(d1(x)) = x)$.
9 $(d1(d(x)) = x)$.
10 $(h(h1(x)) = x)$.
11 $(h1(h(x)) = x)$.
12 $(a(b(c(x))) = d(x))$.
13 $(b(c(d(x))) = h(x))$.
14 $(c(d(h(x))) = a(x))$.
15 $(d(h(a(x))) = b(x))$.
16 $(h(a(b(x))) = c(x))$.

Strategy for Z22

The Knuth-Bendix option was used, with LRPO ($a > a1 > b > b1 > c > c1 > d > d1 > h > h1$). A refutation was not expected; rather we derive a set of equations in which each generator is expressed in terms of $h1$ and $h1^{22}(x) = x$.

Results for Z22

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	398.38	274.19	144.29	418.55	328.30	491.96	400.89	543.02
Generated	2260	2438	2224	4491	3064	5273	4917	6405
Kept	2742	2058	573	501	293	339	310	378
Memory (K)	4885	4502	3767	7502	5522	8587	7305	9892
Gen/sec	5	8	15	10	9	10	12	11
Speedups:								
Proof	1.0	1.5	2.8	1.0	1.2	0.8	1.0	0.7
Gen/sec	1.0	1.6	2.7	1.9	1.6	1.9	2.2	2.1

Notes on Z22

Extracting new demoulators from the K List by symbol count (Section 2.5) caused Roo-1 to perform better than Otter. However, due to extensive back demoduation, Task B was a serious bottleneck in parallel Roo runs, which resulted in very poor speedups.

3.12 Luka-5

This problem is the equational (Wajsberg algebra) version of a problem in the many-valued sentential calculus $L_{\aleph_0}$ of Łukasiewicz[19]. Equalities 2–5 axiomatize the logic, and clause 6 denies an equality that was included in the first axiomatizations of the logic.

Clauses for Luka-5

Input Usable Clauses

1 $(x = x)$.

Input Set of Support Clauses

2 $(i(T, x) = x)$.
3 $(i(i(x, y), i(i(y, z), i(x, z))) = T)$.
4 $(i(i(x, y), y) = i(i(y, x), x))$.
5 $(i(i(n(x), n(y)), i(y, x)) = T)$.
6 $(i(i(i(a, b), i(b, a)), i(b, a)) \neq T)$.

Strategy for Luka-5

The Knuth-Bendix option was used, with LRPO ($i > n > T$). Back subsumption was not applied.

Results for Luka-5

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	4871.68	3596.68	869.33	480.30	317.32	243.71	210.89	180.10
Generated	538329	533705	531348	578189	578895	592728	642889	624213
Kept	2496	2449	2460	2432	2415	2423	2420	2388
Memory (K)	3065	3161	3479	4149	5713	7595	9923	13879
Gen/sec	110	148	611	1203	1824	2432	3048	3465
Speedups from Otter:								
Proof	1.0	1.4	5.6	10.1	15.4	20.0	23.1	27.0
Gen/sec	1.0	1.3	5.6	10.9	16.6	22.1	27.7	31.5
Speedups from Roo-1:								
Proof	0.7	1.0	4.1	7.5	11.3	14.8	17.1	20.0
Gen/sec	0.7	1.0	4.1	8.1	12.3	16.4	20.6	23.4

Notes on Luka-5

The similarity in the number of generated and kept clauses in the Luka-5 searches indicates that the search spaces were similar. The superlinear speedups are thus not explained by the differences in the search spaces. However, the time difference between Otter and Roo-1 hint at an explanation. The statistics in the output files show that Otter spent approximately 1300 more seconds rewriting than Roo-1 did. We have not yet determined the precise reason for the difference.

3.13 JimC-8

This problem was brought to our attention by J. Christian[2], who used it as a benchmark for HIPER, his high-performance Knuth-Bendix completion program. The problem is to find a complete set of reductions for an associative system with 24 left identities and 24 right inverses.

Clauses for JimC-8

Input Set of Support Clauses

1 $(x = x)$.
2 $(f(f(x, y), z) = f(x, f(y, z)))$.
3 $(f(e1, x) = x)$.
$\vdots$
26 $(f(e24, x) = x)$.
27 $(f(x, g1(x)) = e1)$.
$\vdots$
50 $(f(x, g24(x)) = e24)$.

Strategy for JimC-8

The Knuth-Bendix option was used, with LRPO ($g24 > \ldots > g1 > f > e24 > \ldots > e1$, and f with LR status).

Results for JimC-8

	Otter	Roo-1	Roo-4	Roo-8	Roo-12	Roo-16	Roo-20	Roo-24
Run time	778.58	797.51	129.32	110.37	127.78	103.14	139.47	131.36
Generated	37495	37499	37339	48541	72881	54840	67275	75927
Kept	15802	15476	5767	2999	3230	2016	2201	1996
Memory (K)	10793	11879	10951	9447	11940	8617	10885	11780
Gen/sec	48	47	288	439	570	531	482	578
Speedups:								
Run time	1.0	1.0	6.0	7.1	6.1	7.5	5.6	5.9
Gen/sec	1.0	1.0	6.0	9.2	11.9	11.1	10.0	12.0

Notes on JimC-8

Note the great disparity in the amount of work done (clauses generated) in the various runs. Memory use, on the other hand, remains constant.

3.14 Problems Not Included

Some well-known test problems such as Wos-10 and Schubert's Steamroller are not represented in the list of problems, because they are done by Otter in a few seconds and thus are simply not difficult enough to warrant the use of Roo.

4 Stability of Experiments

The nondeterminism introduced by parallelism gives rise to the question of how repeatable the experiments are. It is indeed true that Roo, unlike Otter, does not run the same way every time, even with the same input file and the same number of processes. In this section we report on some experiments to measure, at least on a limited sample of examples, the extent of this phenomenon. We do this by running the very same problem ten times with the same number of processes, and comparing the results.