An Application of Answer Set Programming: Superoptimisation
A Preliminary Report

Martin Brain, Tom Crick, Marina De Vos and **John Fitch**
Department of Computer Science
University of Bath,
Bath BA2 7AY, UK
email: `{mjb,tc,mdv,jpff}@cs.bath.ac.uk`

Abstract

Answer set programming (ASP) is a declarative problem-solving technique that uses the computation of answer set semantics to provide solutions. Despite comprehensive implementations and a strong theoretical basis, ASP has yet to be used for more than a handful of large-scale applications. This paper describes such a large-scale application and presents some preliminary results. The TOAST (Total Optimisation using Answer Set Technology) project seeks to generate optimal machine code for simple, acyclic functions using a technique known as superoptimisation. ASP is used as a scalable computational engine for conducting searches over complex, non-regular domains. The experimental results suggest this is a viable approach to the optimisation problem and demonstrates the value of using parallel answer set solvers.

Introduction

Answer set programming (ASP) is a relatively new technology, with the first computation tools (referred to as answer set solvers) only appearing in the late 1990s (Niemelä & Simons 1997). Initial studies have demonstrated (WASP 2004) that it has great potential in many application areas, including automatic diagnostics (Eiter *et al.* 2000; Nogueira *et al.* 2001), agent behaviour and communication (De Vos *et al.* 2006), security engineering (P. Giorgini & Zannone 2004) and information integration (S. Costantini & Omodeo 2003). However, larger production scale applications are comparatively scarce. One of the few examples of such a system is the *USA-Advisor* decision support system for the NASA Space Shuttle (Nogueira *et al.* 2001). It modelled an extremely complex domain in a concise way; although of great significance to the field it is, in computational terms, relatively small. The only large and difficult programs most answer set solvers have been tested on are synthetic benchmarks. How well do the algorithms and implementations scale? How much memory and how much time is required? This paper makes an initial attempt to answer some of these questions.

This paper investigates the possibility of using ASP technology to generate optimal machine code for simple functions. Modern compilers apply a fixed set of code improvement techniques using a range of approximations rather than aiming to generate optimal code. None of the existing techniques, or approaches to creating new techniques, are likely to change the current state of play.

An approach to obtaining optimal code sequences is called superoptimisation (Massalin 1987). One of the main bottlenecks in this process is the size of the space of possible instruction sequences, with most superoptimising implementations relying on brute force searches to locate candidate sequences and approximate equivalence verification. The TOAST project presents a new approach to the search and verification problems using ASP.

From an ASP perspective, the TOAST project provides a large-scale, real-world application with some programs containing more than a million ground rules. From a compiler optimisation perspective, it might be a step towards tools that can generate truly optimal code, benefiting many areas, especially embedded systems and high performance computing.

This paper presents the results of the first phase of the TOAST project, with the overall infrastructure complete and three machine architectures implemented. We have used off-the-shelf solvers without any domain-specific optimisations, so the results we present also provide useful benchmarks for these answer set solvers.

The rest of this paper is structured as follows: in the next section, we provide a short introduction to modern compiler technology. In two subsections we explain the mechanisms of code optimisation, superoptimisation and verifiable code generation. In a third subsection we investigate the challenges of producing verifiable superoptimised sequences in terms of the length of input code sequences and word length of the target machine. We then give an overview of ASP from a programming language viewpoint. After these two background sections, we introduce the TOAST system and present the preliminary results. The analysis of these results leads to a section detailing the future work of the project.

The Problem Domain

Before describing the TOAST system and how it uses answer set technology, it is important to consider the problem that it seeks to solve and how this fits into the larger field of compiler design.

Compilers and Optimisation

Optimisation, as commonly used in the field of compiler research and implementation, is something of a misnomer.

A typical compiler targeting assembly language or machine code will include an array of code improvement techniques, from the relatively cheap and simple (identification of common sub-expressions and constant folding) (Aho, Sethi, & Ullmann 1986) to the costly and esoteric (auto-vectorisation and inter-function register allocation) (Appel 2004). However, none of these generate optimal code; the code that they output is only improved (though often to a significant degree). As all of these techniques identify and remove certain inefficiencies, it is impossible to guarantee that the code could not be further improved.

Further confusion is created by complications in defining optimality. In the linear case, a shorter instruction sequence is clearly better[1]. If the code branches but is not cyclic, a number of definitions are possible: shortest average path, shortest over all sequence, etc. However, for code including cycles, it is not possible to define optimality in the general case. To do so would require calculating how many time the body of loop would be executed – a problem equivalent to the halting problem. To avoid this, and problems with other areas such as equivalence of floating point operations, this paper only considers optimality in terms of the number of instructions used in acyclic, integer-based code.

Finally, it is important to consider the scale of the likely savings. The effect of improvements in code generation for an average program have been estimated as a 4% speed increase[2] per year (Proebsting 1998). In this context, saving just one or two instructions is significant, particularly if the technique is widely applicable, or can be used to target 'hot spots', CPU-intensive sections of code.

Superoptimisation

Superoptimisation is a radically different approach to code generation, first described in (Massalin 1987). Rather than starting with crudely generated code and improving it, a superoptimiser starts with the specification of a function and performs an exhaustive search for a sequence of instructions that meets this specification. Clearly, as the length of the sequence increases, the search space potentially rises at an exponential rate. This makes the technique unsuitable for use in normal compilers, but for improving the code generators of compilers and for targeting key sections of performance-critical functions, the results can be quite impressive.

A good example of superoptimisation is the sign function (Massalin 1987), which returns the sign of a binary integer, or zero if the input is zero:

```
int signum (int x) {
        if (x > 0)      return 1;
        else if (x < 0) return -1;
        else            return 0;
}
```

A naïve compilation of this function would produce approximately ten instructions, including at least two conditional branch instructions. A skilled assembly language programmer may manage to implement it in four instructions with one conditional branch. At the time of writing, this is the best that state of the art compilation can produce. However, superoptimisation (in this case for the SPARC-V7 architecture) gives the following:

```
! input in %i0
addcc   %i0 %i0 %l1
subxcc  %i0 %l1 %l2
addx    %l2 %i0 %o1
! output in %o1
```

Not only is this sequence only three instructions long, it does not require any conditional branches, a significant saving on modern pipelined processors. This example also demonstrates another interesting property of code produced by superoptimisation: it is not obvious that this computes the sign of a number or how it does so. The pattern of addition and subtraction essentially 'cancels out', with the actual computation done by how the carry flag is set and used by each instruction (instructions whose name includes `cc` set the carry flag, whereas instructions with `x` use the carry flag). Such inventive use of a processor's features are common in superoptimised sequences; when the GNU Superoptimizer (GSO) (Granlund & Kenner 1992) was first used to superoptimise sequences for the GCC port to the POWER architecture, it produced a number of sequences that were shorter than the processor's designers thought possible!

Despite significant potential, superoptimisation has received relatively little attention within the field of compiler research. Following Massalin's work, the next published superoptimiser was GSO, a portable superoptimiser developed to aid the development of GCC. It improved on Massalin's search strategy by attempting to apply constraints while generating elements of the search space, rather than generating all possible sequences and then skipping those that were marked as clearly redundant. The most recent work on superoptimisation have been from the Denali project (Joshi, Nelson, & Randall 2002; Joshi, Nelson, & Zhou 2003). Their approach was much closer to that of the TOAST system, using automatic theorem-proving technology to handle the large search spaces.

Analysis of Problem Domain

Superoptimisation naturally breaks into two sub-problems: searching for sequences that meet some limited criteria and verifying which of these candidates are fully equivalent to the input function.

The search space of possible sequences of a given length is very large, at least the number of instructions available to the power of the length of the sequences (thus growing at least exponentially as the length rises). However, a number of complex constraints exist that reduce the space that has

[1]Although the TOAST approach could be generalised to handle them, this paper ignores complications such as pipelining, caching, pre-fetching, variable-instruction latency and super-scalar execution.

[2]This may seem very low in comparison with the increase in processing power created by advances in microprocessor manufacturing. However, it is wise to consider the vast disparity in research spending in the two areas, as well as the link between them: most modern processors would not achieve such drastic improvements without advanced compilers to generate effi cient code for them.

to be searched. For example, if a sub-sequence is known to be non-optimal then anything that includes it will also be non-optimal and thus can be discarded. Managing the size and complexity of this space is the current limit on superoptimiser performance.

Verifying that two code sequences are equivalent also involves a large space of possibilities (for single input sequences it is 2^w where w is the word length (number of bits per register) of the processor). However, it is a space that has a number of unusual properties. Firstly, verification of two sequences is a reasonably simple task for human experts, suggesting there may be a strong set of heuristics. Secondly, sequences of instructions that are equivalent on a reasonably small subset of the space of possible inputs tend to be equivalent on all of it. Both GSO and Massalin's original superoptimiser handled verification by testing the new sequence for correctness of a small number of inputs and declaring it equivalent if it passed. Although non-rigorous, this approach seemed to work in practise (Granlund & Kenner 1992).

Answer Set Programming

Answer set programming is a declarative problem solving technique based on research on the semantics of logic programming languages and non-monotonic reasoning (Gelfond & Lifschitz 1988; 1991). For reasons of compactness, this paper only includes a brief summary of answer set semantics; a more in-depth discussion can be found in (Baral 2003).

Answer set semantics are defined with respect to *programs*, sets of Horn clause-style rules composed of literals. Two forms of negation are described, negation as failure and explicit (or classical) negation. The first (denoted as not) is interpreted as not knowing that the literal is true, while the second (denoted as $\neg$) is knowing that the literal is not true. For example:

$$\begin{array}{rcl} a & \leftarrow & b, not\ c. \\ \neg b & \leftarrow & not\ a. \end{array}$$

is interpreted as "a is known to be true if b is known to be true and c is not known to be true. b is known to be not true if a is not known to be true" (the precise declarative meaning is an area of ongoing work, see (Denecker 2004)). Constraints are also supported, which allow conjunctions of literals to be ruled as inconsistent. Answer sets are sets of literals that are consistent (do not contain both a and $\neg a$ or the bodies of any constraints) and supported (every literal has at least one, acyclic way of concluding its truth). A given program may have zero or more answer sets.

Answer set programming is describing a problem as a program under answer set semantics in such a way that the answer sets of the program correspond to the solutions of the problem. In many cases, this is simply a case of encoding the description of the problem domain and the description of what constitutes a solution. Thus solving the problem is reduced to computing the answer sets of the program.

Computing an answer set of a program is an NP-complete task, but there are a number of sophisticated tools, known as answer set solvers, that can perform this computation. The first generation of efficient solvers (such as SMODELS (Niemelä & Simons 1997) and DLV (Leone *et al.* 2006)) use a DPLL-style algorithm (Davis, Logemann, & Loveland 1962). Before computation, the answer set program is *grounded* (an instantiation process that creates copies of the rules for each usable value of each variable) by using tools such as LPARSE (Syrjänen 2000), to remove variables. The answer sets are then computed using a backtracking algorithm; at each stage the sets of literals that are known to be true and known to be false are expanded according to a set of simple rules (similar to unit propagation in DPLL), then a branching literal is chosen according to heuristics and both possible branches (asserting the literal to be true or false) are explored. An alternative approach is to use a SAT solver to generate candidate answer sets and then check whether these meet all criteria. This is the approach used by CMODELS (Giunchiglia, Lierler, & Maratea 2004). More recent work has investigated using 'Beowulf'-style parallel systems to explore possible models in parallel (Pontelli, Balduccini, & Bermudez 2003). One such system, PLATYPUS (Gressmann *et al.* 2005) is used in the TOAST system.

TOAST

The existence of a clear NP algorithm, as well as the causal nature of the problem and the need for high expressive and computational power, suggest ASP as a suitable approach to the superoptimisation problem. The TOAST system consists of a number of components that generate answer set programs and parse answer sets, with a 'front end' that uses these components to produce a superoptimised version of an input function. Data is passed between components either as fragments of answer set programs or in an architecture-independent, assembly language-like format. An answer set solver is used as a 'black box' tool, currently either SMODELS or PLATYPUS, although experiments with other solvers are ongoing. Although the grounding tool of DLV is stronger in some notable examples, it has not been tested yet due to syntax incompatibilities with many of the features required.

System Components

Four key components provide most of the functionality of the TOAST system:

pickVectors Given the specification of the input to an instruction sequence, *pickVectors* creates a representative set of inputs, known as input vectors, and outputs it as an ASP program fragment.

execute This component takes an ASP program fragment describing an input vector (as generated by *pickVector* or *verify*) and emulates running an instruction sequence with that input. The output is the given as another ASP program fragment containing constraints on the instruction sequence's outputs.

search Taking ASP fragments giving 'input' and 'output' values (from *pickVectors* / *verify* and *execute* respectively), this component searches for all instruction sequences of a given length that produce the required 'output' for the given 'input' values.

verify Takes two instruction sequences with the same input specification and tests if they are equivalent. If they are not, an input vector on which they differ can be generated, in the format used by *execute* and *search*.

The TOAST system is fully architecture-independent. Architecture-specific information is stored in a description file which provides meta-information about the architecture, as well as which operations from the library of instructions are available. At the time of writing, TOAST supports the MIPS R2000 and SPARC V7/V8 processors. Porting to a new architecture is simple and takes between a few hours and a week, depending on how many of the instructions have already been modelled.

System Architecture

The key observation underlying the design of the TOAST system is that any correct superoptimised sequence will be returned by running *search* for the appropriate instruction length; however, not everything that *search* returns is necessarily a correct answer. Thus to generate superoptimised sequences, the front end uses *pickVector* and *execute* on the input instruction sequence to create criteria for *search*. Instruction sequence lengths from one up to one less than the length of the original input sequence are then sequentially searched. If answers are generated, another set of criteria are created and the same length searched again. The two sets are then intersected, as any correct answer must appear in both sets. This process is repeated until either the intersection becomes empty, in which case the search moves on to the next sequence length, or until the intersection does not decrease in size. *verify* can then be used to check members of this set for equivalence to the original input program.

The Answer Set Programs

In the following section we give a brief overview of the basic categories of answer set programs generated within the system: flow control, flag control, instruction sequences, instruction definitions, input vectors and output constraints.

The flow control rules set which instruction will be 'executed' at a given time step by controlling the `pc` (program counter) literal. An example set of flow control rules are given in Figure 1. The rules are simple, such as an instruction that asserts `jump(C,T,J)` would move the program's execution on `J` instructions, otherwise it will just move on by one. As the ASP programs may need to simultaneously model multiple independent code streams (for example, when trying to verify their equivalence), all literals are tagged with a abstract entity called 'colour'. The inclusion of the `colour(C)` literal in each rule allows copies to be created for each separate code stream during instantiation. In most cases, when only one code stream is used, only one value of `colour` is defined and only one copy of each set of rules is produced; the overhead involved is negligible.

Flag control rules control the setting and maintenance of processor flags such as carry, overflow, zero and negative. Although generally only used for controlling conditional

```
haveJumped(C,T) :- jump(C,T,J), jumpSize(C,J),
                   time(C,T), colour(C).

pc(C,PCV+J,T+1) :- pc(C,PCV,T), jump(C,T,J), jumpSize(C,J),
                   time(C,T), colour(C), position(C,PCV).

pc(C,PCV+1,T+1) :- pc(C,PCV,T), not haveJumped(C,T),
                   time(C,T), colour(C), position(C,PCV).

pc(C,1,1).
```

Figure 1: Flow Control Rules in ASP

```
value(C,T,B) :- istream(C,P,lxor,R1,R2,none), pc(C,P,T),
                value(C,R1,B), -value(C,R2,B),
                register(R1), register(R2), colour(C),
                position(C,P), time(C,T), bit(B).

value(C,T,B) :- istream(C,P,lxor,R1,R2,none), pc(C,P,T),
                -value(C,R1,B), value(C,R2,B),
                register(R1), register(R2), colour(C),
                position(C,P), time(C,T), bit(B).

-value(C,T,B) :- istream(C,P,lxor,R1,R2,none), pc(C,P,T),
                 not value(C,T,B), register(R1), register(R2),
                 colour(C), position(C,P), time(C,T), bit(B).

symmetricInstruction(lxor).
```

Figure 2: Modelling of a Logical XOR Instruction in ASP

branches and multi-word arithmetic, the flags are a source of many superoptimised sequences and are thus of prime importance when modelling.

The instruction sequence itself is represented as a series of facts, or in the case of *search*, a set of choice rules (choice rules are a syntactic extension to ASP, see (Niemelä & Simons 1997)). The literals are then used by the instruction definitions to control the `value` literals that give the value of various registers within the processor. If the literal is in the answer set, the given bit is taken to be a 1, if the classically-negated version of the literal is in the answer set then it is a 0. An example instruction definition, for a logical XOR (exclusive or) between registers, is given in Figure 2. Note the use of negation as failure to reduce the number of rules required and the declaration that `lxor` is symmetric, which is used to reduce the search space.

The input vectors and output constraints are the program fragments created by *pickVectors* and *execute* respectively.

The ASP programs generated do not contain disjunction, aggregates or any other non-syntactic extensions to answer set semantics.

Results

Tests were run on a Beowulf-style cluster of 20 x 800MHz Intel Celeron, 512MB RAM machines connected by 100Mb Ethernet, running SuSE Linux 9.2. Results are given for SMODELS v2.28 (denoted s) and the initial MPI version of PLATYPUS running on n nodes (denoted p/n). LPARSE v1.0.17 was used in all cases to ground the programs. The timings displayed are from the SMODELS internal timing mechanism and the PLATYPUS MPI wall time respectively. Values for LPARSE are the user times given via the system `time` command.

Search Time

search was used to generate programs that searched the space of SPARC-V7 instructions for candidate superoptimisations for the following instruction sequence:

```
! input in %i0, %i1
and     %i0 %i1 %l1
add     %i0 %l1 %l2
add     %i0 %l2 %l3
sub     0   %l3 %o1
! output in %o1
```

This sequence was selected as a ‘worst case’, an example of a sequence that cannot be superoptimised, giving an approximate ceiling on the performance of the system.

Statistics on the programs used can be found in Figure 3, with the timing results are given in Figure 4.

Verification Time

verify was used to create a verification program for the following two code sequences:

```
! input in %i0      ! input in %i0
add %i0 %i0 %o1     umult %i0  2  %o1
! output in %o1     ! output in %o1
```

using the SPARC-V8[3] architecture but varying the processor word length (the number of bits per register). This pair of programs were chosen as, although they are clearly equivalent, the modelling and reasoning required to show this is non-trivial. Timing results for a variety of solver configurations and different word lengths can be found in Figure 7, program statistics can be found in Figure 5.

Analysis

The experimental results presented suggest a number of interesting points. Firstly, superoptimisation using ASP is feasible, but work is needed to make it more practical. Given that only a few constraints were used in the programs generated by *search*, increasing the length of the maximum practical search space seems eminently possible. The result from *verify* are less encouraging; although it shows it is possible using ASP, it also suggests that attempting to verify instruction sequences of more than 32 bits of input is likely to require significant resources.

The graph in Figure 6 also shows some interesting properties of the parallel solver. The overhead of the solver appears to be near constant, regardless of the number of processors used. For the simpler problems, the overhead of the parallel solver is greater than any advantages, but for the larger problems it makes a significant difference and the speed-up is approximately proportional to the number of processors used.

Finally, the figures suggest that the SMODELS algorithm does not scale linearly on some programs. The programs output by *verify* double in search space size for each increase in word length, but the time required by SMODELS rises by significantly more than a factor of two. Strangely, this additional overhead appears to be less significant as the number of processors used by PLATYPUS rises.

[3]SPARC-V8 is a later, minimal extension of SPARC-V7 with the addition of the `umult` instruction.

The simplified graph in Figure 6 shows these effects, with time graphs for SMODELS against PLATYPUS with 4, 8 and 16 processors.

Future Development

One of the key targets in the development of TOAST is to reduce the amount of time required in searching. Doing so will also increase the length of instruction sequence that can be found. This requires improvements to both the programs that are generated and the tools used to solve them.

A key improvement to the generated programs will be to remove all short sequences that are known to be non-optimal. *search* can be used to generate all possible instruction sequences of a given length. By superoptimising each one of these for the smaller lengths, it is then possible to build a set of equivalence categories of instructions. Only the shortest member of each category needs to be in the search space and thus a set of constraints can be added to the programs that *search* generates. This process only ever needs to be done once for each processor architecture and will give significant improvements in terms of search times. The equivalence classes generated may also be useful to improve verification.

The other developments needed to reduce the *search* time are in the tools used. Addressing the amount of memory consumed by LPARSE and attempting to improve the scaling of the SMODELS algorithm are both high priorities.

The performance of *verify* also raises some interesting questions. At present, is possible to verify programs for some of the smaller, embedded processors. However, in its current form it is unlikely to scale to high-end, 64 bit processors. A number of alternative approaches are being considered, such as attempting to prove equivalence results about the generated ASP programs, reducing the instructions to a minimal/pseudo-normal form (an approach first used by Massalin), using some form of algebraic theorem-proving (as in the Denali project) or attempting to formalise and prove the observation that sequences equivalent on a small set of points tend to be equivalent on all of them.

Using the TOAST system to improve the code generated by tools such as GCC is also a key target for the project. By implementing tools that translate between the TOAST internal assembly-like format and processor-specific assembly, it will be possible to check the output of GCC for sequences that can be superoptimised. Patterns that occur regularly can then be added to the instruction generation phases of GCC. Performance-critical system libraries, such as the GNU Multiple Precision Arithmetic Library (GMP) (Granlund 2006) and code generators used by Just In Time (JIT) compilers could also be interesting application areas.

It is hoped that it will not only prove useful as a tool for optimising sections of performance critical code, but that the ASP programs could be used as benchmarks for solver performance and the basis of other applications which reason about machine code.

Length of Sequence	No. rules	Grounding time	No. ground rules	No. of atoms
1	530	20.100	95938	1018
2	534	65.740	298312	1993
3	538	142.22	643070	3428
4	542	-	1197182	6873

Figure 3: Search Program Sizes

Length of Sequence	s	$p/2$	$p/4$	$p/6$	$p/8$	$p/10$	$p/12$	$p/14$	$p/16$	$p/18$	$p/20$
1	3.057	10.4647	10.4813	10.4761	10.5232	10.5023	10.4674	10.4782	10.4833	10.4915	10.5040
2	99.908	104.710	123.312	120.984	135.733	136.057	139.944	139.000	135.539	139.271	138.288
3	81763.9	63644.4	19433.4	12641.0	6008.20	7972.73	9097.83	6608.64	6063.08	4629.90	5419.08
4	$>$ 237337.35	-	-	-	-	-	-	-	-	-	-

Figure 4: Search Space Size v Compute Time (secs)

Word Length	No. rules	Grounding time	No. ground rules	No. of atoms
8	779	1.220	1755	975
9	780	1.320	2063	1099
10	781	1.430	2402	1235
11	782	1.480	2772	1383
12	783	1.330	3173	1543
13	784	1.350	3605	1715
14	785	1.450	4068	1899
15	786	1.480	4562	2095
16	787	1.480	5087	2303
17	788	1.640	5645	2527
18	789	1.680	6234	2763
19	790	1.690	6854	3011
20	791	1.550	7505	3271
21	792	1.590	8187	3543
22	793	1.670	8900	3827
23	794	1.900	9644	4123
24	795	1.830	10419	4431

Figure 5: Verification Program Sizes

Conclusion

This paper suggests that ASP can be used to solve large-scale, real-world problems. Future work will hopefully show this is also a powerful approach to the superoptimisation problem and perhaps even a 'killer application' for ASP.

However, it is not without challenges. Although savings to both size of the ASP programs used and their search spaces are possible, this will remain a high-end application for answer set solvers. Some of the features required, such as the handling of large, sparse search spaces and efficiency in producing all possible answer sets (or traversing the search space of programs without answer sets) are unfortunately not key targets of current solver development.

The TOAST project demonstrates that answer set technology is ready to be used in large-scale applications, although more work is required to make it competitive.

References

Aho, A. V.; Sethi, R.; and Ullmann, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Appel, A. W. 2004. *Modern Compiler Implementation in C*. Cambridge University Press.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A Machine Program for Theorem-Proving. *Communications of the ACM* 5(7):394–397.

De Vos, M.; Crick, T.; Padget, J.; Brain, M.; Cliffe, O.; and Needham, J. 2006. A Multi-agent Platform using Ordered Choice Logic Programming. In *Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT'05)*, volume 3904 of *LNAI*, 72–88. Springer.

Denecker, M. 2004. What's in a Model? Epistemological Analysis of Logic Programming. In *Proceedings of the 9th*

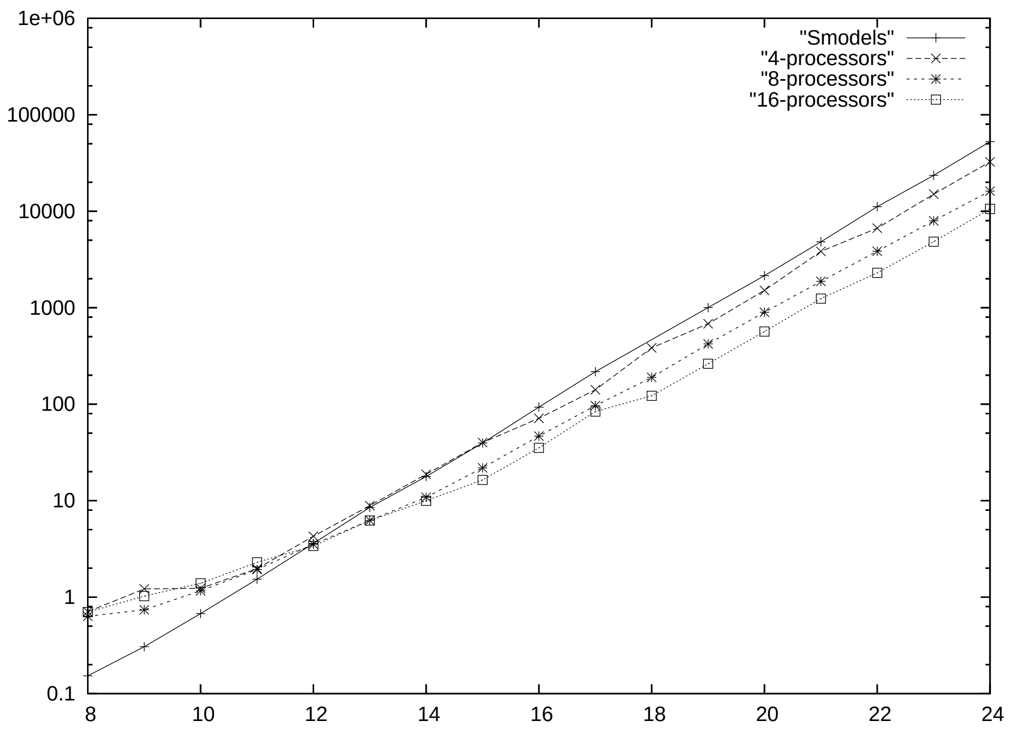

Figure 6: Simplified Timings (Log Scale)

Word Length	s	$p/2$	$p/4$	$p/6$	$p/8$	$p/10$	$p/12$	$p/14$	$p/16$	$p/18$	$p/20$
8 bit	0.153	0.495074	1.21623	0.581861	0.632791	0.662914	0.706752	1.21751	0.698032	0.723088	0.740474
9 bit	0.306	0.863785	0.705636	0.777568	0.740043	1.02031	0.918548	0.864449	1.02644	1.03752	1.09627
10 bit	0.675	1.61512	1.2337	1.23213	1.16333	1.23683	1.28347	1.28118	1.39326	1.29568	1.31185
11 bit	1.537	3.42153	1.97181	1.84315	1.93191	2.01146	1.9929	2.34911	2.2948	2.28081	2.18609
12 bit	3.597	7.46042	4.28284	3.43396	3.53243	3.33475	3.27878	3.16487	3.38788	3.21397	3.94176
13 bit	8.505	15.8174	8.86814	6.51479	6.25371	5.55683	5.1507	5.3369	6.22179	5.61428	5.06376
14 bit	17.795	34.2229	18.7478	15.9874	10.8228	9.57001	9.3808	8.6161	9.97594	9.41512	8.16737
15 bit	39.651	76.018	39.9688	25.9992	21.8607	19.382	17.6372	18.0614	16.3806	15.6143	15.6043
16 bit	93.141	167.222	71.3785	52.7732	46.6144	36.5995	31.9568	33.2825	35.3159	27.2188	29.5464
17 bit	217.162	373.258	141.108	110.65	96.6821	85.1217	77.4811	78.7892	83.9177	56.1338	58.4057
18 bit	463.025	815.373	384.237	222.826	189.690	162.318	144.840	136.126	122.038	118.658	133.579
19 bit	1002.696	1738.02	681.673	456.607	421.681	430.879	299.870	290.456	262.611	229.802	217.998
20 bit	2146.941	3790.84	1514.80	994.849	896.705	726.629	625.820	610.117	566.040	523.700	426.004
21 bit	4826.837	8206.4	3438.71	2279.3	1874.36	1544.74	1461.4	1199.96	1244.95	932.877	1128.53
22 bit	11168.818	17974.8	6683.06	4375.12	3850.71	3017.14	3206.33	2492.00	2296.87	2245.3	1869.17
23 bit	23547.807	38870.5	15047	9217.82	7947.95	7123.56	6111.6	6089.38	4833.66	4610.92	4020.37
24 bit	52681.498	83405.1	32561.2	20789.1	16165.4	14453.8	12800.7	11213.2	10580.4	9199.8	8685.47

Figure 7: Word Length v Compute Time (secs)

International Conference on the Principles of Knowledge Representation and Reasoning (KR2004), 106–113.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2000. Using the dlv system for planning and diagnostic reasoning. In *Proceedings of the 14th Workshop on Logic Programming (WLP'99)*, 125–134.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Kowalski, R. A., and Bowen, K., eds., *Proceedings of the 5th International Conference on Logic Programming (ICLP'88)*, 1070–1080. The MIT Press.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9(3-4):365–386.

Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2004. SAT-Based Answer Set Programming. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-04)*, 61–66.

Granlund, T., and Kenner, R. 1992. Eliminating Branches using a Superoptimizer and the GNU C Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*,

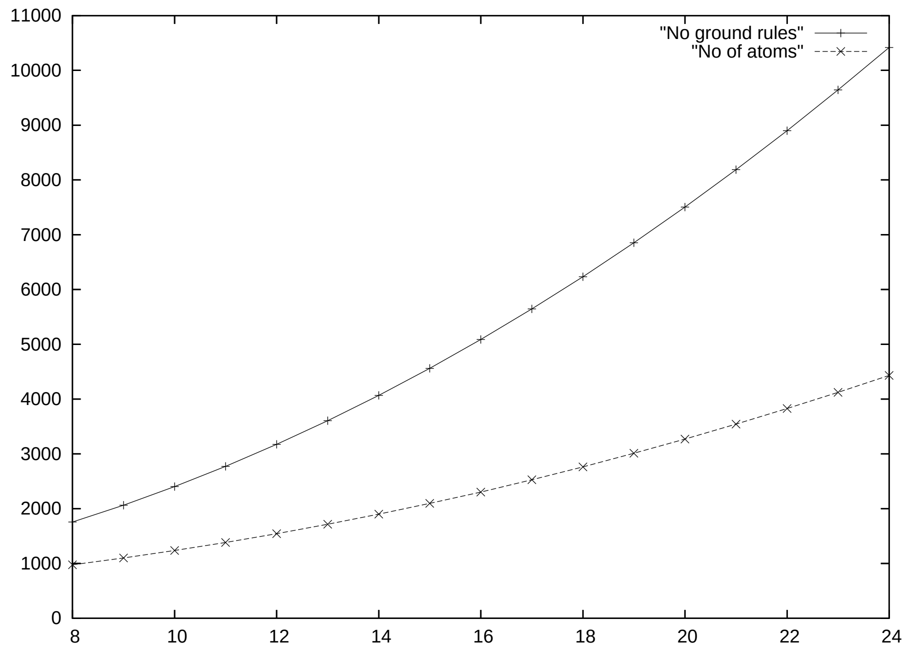

Figure 8: Number of Rules/Atoms v Word Length

341–352. ACM Press.

Granlund, T. 2006. GMP : GNU Multiple Precision Arithmetic Library. `http://www.swox.com/gmp/`.

Gressmann, J.; Janhunen, T.; Mercer, R.; Schaub, T.; Thiele, S.; and Tichy, R. 2005. Platypus: A Platform for Distributed Answer Set Solving. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, 227–239.

Joshi, R.; Nelson, G.; and Randall, K. 2002. Denali: A Goal-Directed Superoptimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, 304–314. ACM Press.

Joshi, R.; Nelson, G.; and Zhou, Y. 2003. The Straight-Line Automatic Programming Problem. Technical Report HPL-2003-236, HP Labs.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for Knowledge Representation and Reasoning. *to appear in ACM Transactions on Computational Logic (TOCL)*.

Massalin, H. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'87)*, 122–126. IEEE Computer Society Press.

Niemelä, I., and Simons, P. 1997. Smodels: An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *LNAI*, 420–429. Springer.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog Decision Support System for the Space Shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, 169–183. Springer-Verlag.

P. Giorgini, F. Massacci, J. M., and Zannone, N. 2004. Requirements Engineering Meets Trust Management: Model, Methodology, and Reasoning. In *Proceedings of the 2nd International Conference on Trust Management (iTrust 2004)*, volume 2995 of *LNCS*, 176–190. Springer.

Pontelli, E.; Balduccini, M.; and Bermudez, F. 2003. Non-Monotonic Reasoning on Beowulf Platforms. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, 37–57. Springer-Verlag.

Proebsting, T. 1998. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. `http://research.microsoft.com/~toddpro/papers/law.htm`.

S. Costantini, A. F., and Omodeo, E. 2003. Mapping Between Domain Models in Answer Set Programming. *Proceedings of Answer Set Programming: Advances in Theory and Implementation (ASP'03)*.

Syrjänen, T. 2000. *Lparse 1.0 User's Manual*. Helsinki University of Technology.

WASP. 2004. WP5 Report: Model Applications and Proofs-of-Concept. `http://www.kr.tuwien.ac.at/projects/WASP/wasp-wp5-web.html`.