TOAST: Applying Answer Set Programming to Superoptimisation

Martin Brain, Tom Crick, Marina De Vos and John Fitch

Department of Computer Science
University of Bath
Bath BA2 7AY, UK
{mjb,tc,mdv,jpff}@cs.bath.ac.uk

Abstract. Answer set programming (ASP) is a form of declarative programming particularly suited to difficult combinatorial search problems. However, it has yet to be used for more than a handful of large-scale applications, which are needed to demonstrate the strengths of ASP and to motivate the development of tools and methodology. This paper describes such a large-scale application, the TOAST (Total Optimisation using Answer Set Technology) system, which seeks to generate optimal machine code for simple, acyclic functions using a technique known as superoptimisation. ASP is used as a scalable computational engine to handle searching over complex, non-regular search spaces, with the experimental results suggesting that this is a viable approach to the optimisation problem and demonstrates the scalability of a variety of solvers.

1 Introduction

Answer set programming (ASP) is a relative new technology, with the first computation tools only appearing in the late 1990s [1, 2]. Initial studies have demonstrated [3] that it has potential in many areas, including automatic diagnostics [4, 5], agent behaviour and communication [6], security engineering [7] and information integration [8]. However, larger production-scale applications are comparatively scarce. One of the few examples of such a system is the *USA-Advisor* decision support system [5] for the NASA Space Shuttle. It modelled a 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; thus it is not yet known how well these algorithms and implementations scale.

This paper investigates the possibility of using ASP technology to generate optimal machine code for simple functions. Modern compilers only provide code improvements via a range of approximations rather than generating truly optimal code; none of these existing techniques, or approaches to creating new techniques, are likely to change the current state of play.

An approach to generating optimal code sequences is called superoptimisation [9]. 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 then approximate equivalence verification.

From an ASP perspective, the TOAST project provides a large-scale, real-world application, with certain programs containing more than a million ground rules. From a compiler perspective, it might be a step towards tools that can generate truly optimal code, which could have an impact on many areas, especially resource-critical environments such as embedded systems and high performance computing.

This paper presents the results of the first phase of the TOAST project, with the infrastructure complete and three machine architectures implemented. At present, off-the-shelf solvers are used without any domain-specific optimisations, so the results we present provide not only useful benchmarks for TOAST, but also for the answer set solvers themselves.

2 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 solves and how this fits into the larger field of compiler design.

2.1 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 (such as identification of common sub-expressions and constant folding) [10] to the costly and esoteric (such as auto-vectorisation and inter-function register allocation) [11]. However, none of these generate optimal code; the code that they output is only improved (though often to a significant degree). All of these techniques identify and remove specific inefficiencies, but it is impossible to guarantee that the code could not be further improved.

Further confusion between code improvement and the generation of optimal code is created by complications in defining optimality. In the linear case, a shorter instruction sequence is quite clearly better[1]. However, if the code branches, but is acyclic, a number of definitions are possible: shortest average path, shortest over all sequences, etc. For code containing cycles, it is not possible to define optimality in the general case. To do so would require calculating how many times the body of the loop would be executed – a problem equivalent to the halting problem. To avoid this and other problem 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.

It is also important to consider the scale of savings that are likely to be achieved. The effect of improvements in code generation for an average program have been estimated as a 4% speed increase per year[2] [12]. In this context, a saving of just one or two

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

[2] This may seem very low in comparison with the increase in processing power created by advances in processor manufacture. However, it is wise to consider the huge disparity in research

instructions is significant, particularly if the technique is widely applicable or can be used to target 'hot spots', CPU-intensive sections of code.

2.2 Superoptimisation

Superoptimisation is a radically different approach to code generation, first described by Massalin [9]. 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 increases at an exponential rate. This makes the technique unsuitable for use in normal compiler toolchains, though for improving the code generators of compilers and for targeting key sections of performance critical functions, the results can be impressive.

A good example of superoptimisation is demonstrated by the sign function [9], 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 simple compilation of this function would generate ten or so instructions, including at least two conditional branch instructions. A skilled assembler programmer might manage to implement it in four instructions with only one conditional branch. Current state of the art compilers can normally achieve the same. However, superoptimisation of this function (here for the SPARC-V7 architecture) gives the following sequence of three instructions:

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

Not only is this sequence shorter, it does not require any conditional branches, a significant saving on modern pipelined processors. This example shows 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 'cancels out', with the actual computation being done by how the carry flag is set and used by each instruction (instructions whose names include `cc` set the carry flag, likewise `x` denotes instructions that use the carry flag). Such inventive uses of the processor's features are common in superoptimised sequences; when the GNU Superoptimizer (GSO) [13] was 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!

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 efficient code for them.

Despite significant potential, superoptimisation has received relatively little attention within the field of compiler research. Following Massalin's work, the next superoptimising implementation was GSO, a portable superoptimiser developed to improve the GCC toolchain. It advanced 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 [14, 15]. Their approach was much closer to that of the TOAST project, using automatic theorem proving technology as an 'intelligent' approach to handling the large search spaces.

2.3 Analysis of Problem Domain

Superoptimisation naturally decomposes into two sub-problems: searching for sequences that meet some limited criteria and then 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 constraints exist that reduce the amount of the space that has 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. Handling 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 a sequence with a single input it is 2^w, where w is the word length (the number of bits per register) of the machine architecture). However, it is a space that shows a number of unusual properties. Firstly, the process is a reasonably simple task for human experts, suggesting there may be a set of strong 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 Massalin's superoptimiser and GSO handled verification by testing the new sequence for correctness on a representative set of inputs and declaring it equivalent if it passed. Although non-rigorous, this approach seemed to work in practise [9, 13].

Due to the problems in producing accurate, formal models of the search space of instruction sequences the complexity of superoptimisation is currently unknown.

3 Answer Set Programming

Answer set programming is a form of declarative programming oriented towards solving difficult combinatorial search problems, which has emerged from research on the semantics of logic programming languages and non-monotonic reasoning [16, 17]. The answer set semantics have developed into an intuitive system for 'real-world reasoning'. For reasons of compactness, this paper only includes a brief summary of answer set semantics, though a more in-depth description can be found in [18].

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:

$$a \leftarrow b, not\ c.$$
$$\neg b \leftarrow not\ a.$$

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 [19] for a further discussion). Constraints are also supported, which allow conjunctions of literals to be deemed inconsistent. Answer sets are sets of literals that are consistent (i.e. do not contain both a or $\neg a$), not constrained (do not contain 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 the 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 [1] and DLV [20]) use a DPLL-style [21] algorithm. 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 [22], 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 not known to be true are expanded according to 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 known to be true or not) 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 [23] and ASSAT [24]. More recent work has investigated using 'Beowulf'-style parallel systems to explore possible models in parallel [25].

4 Total Optimisation using Answer Set Technology

The TOAST system consists of a number of components that generate answer set programs and parse answer sets, with a 'front end' which 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 then used as a 'black box' tool (currently any solver that supports LPARSE syntax).

ASP was chosen because the structure of the rules simplifies the modelling of the bitwise operation of instructions, while also allowing the modelling of complex constraints.

4.1 System Components

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

pickVectors
input: variable specification
output: ASP vectors
Given the specification of the inputs to an instruction sequence, a representative set of inputs (known as input vectors) are generated in ASP.

execute
input: ASP vectors, program
output: ASP constraints
Takes the input vectors (as generated by *pickVectors* or *verify*) and emulates running an instruction sequence with that input, giving constraints on the instruction sequence's outputs.

search
input: search space, ASP vectors, ASP constraints
output: program fragments
Takes ASP fragments giving 'start' and 'end' values (from *pickVectors*/*verify* and *execute* respectively) and searches for all instruction sequences of a given length (the search space) that produce the correct output with the described input.

verify
input: program, program
output: boolean
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 output, in a suitable form for *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 following architectures: MIPS [26], SPARC-V7 and SPARC-V8 [27]. Porting to a new architecture is simple and takes between a few hours and a week, depending on how many of the instructions used have already been modelled.

4.2 System Architecture

The key observation underlying the design of the TOAST system is that any superoptimised sequence will necessarily be returned by using *search* on 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 input sequence, are then searched sequentially. If any answers are given, another criteria set is created and the same length searched again. The two sets are then intersected, as any correct answer

```
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).
```

Fig. 1. Flow Control Rules in ASP

must appear in both. This process is repeated until either the intersection is empty, in which case the search moves on to the next length, or until the intersection stops getting any smaller. *verify* can then be used to check members of this set for equivalence to the original input program.

4.3 The Answer Set Programs

All of the answer sets programs created in the system consist of a number of basic components.

Flow control rules define 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 fairly self-explanatory, for example, an instruction that asserts `jump(C,T,J)`, moves the program's execution on `J` instructions, otherwise it will just move forward by one. As ASP programs may need to simultaneously model multiple independent code streams (for example, when trying to verify their equivalence), all literals are 'tagged' with an abstract property called colour. The inclusion of the `colour(C)` literal in each rule then allows copies to be created for each 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 model the setting and maintenance of processor flags such as carry, overflow, zero and negative. Although generally only used for controlling conditional branches and multi-word arithmetic, these flags are a source of many superoptimised sequences and are thus of prime importance when modelling a processor.

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 [1]). These 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 needed and the declaration that `lxor` is symmetric, which is used to reduce the search space.

Input vectors and *output constraints* are the program fragments created by *pickVectors* and *execute* respectively. None of the ASP programs generated require disjunction, aggregates or any other non-syntactic extensions to answer set semantics.

```
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).
```

Fig. 2. Modelling of a Logical XOR Instruction in ASP

5 Testing

In this section we present preliminary results on using ASP as a tool for superoptimisation, also showing some interesting properties of the different answer set solvers used in the tests.

5.1 System and Solvers

Tests were run on a Beowulf-style cluster of sixteen 800MHz Intel Celeron, 512MB RAM machines connected by 100Mb Ethernet, running SuSE Linux 9.2. Results are given for SMODELS-2.28 (denoted s), SURYA [28] (denoted u), NOMORE++ 1.4 [29] (denoted m) and the MPI version of PLATYPUS running on n nodes (denoted p/n). CMODELS, ASSAT and LPSM were also tested, though concerns over their stability and correctness means the results are not presented. DLV has yet to be tested, as the difference in syntax would require extensive alterations to the input programs. It is hoped that we will soon be able to publish results for these solvers. LPARSE-1.0.17 was used to ground the programs. Times for the sequential solvers were recorded using the system `time` command, while the MPI wall time was used for PLATYPUS (both given in seconds).

5.2 Test Programs

Three suites of programs were used in the tests. In the first, *search* was used to generate programs that searched the space of SPARC-V7 instructions for candidate optimisations 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. Programs `s1` to `s4` are searches over the spaces of 1 to 4 instructions respectively.

The remaining two test suites give different approaches to testing *verify*. In the first case, an older encoding of the search space using choice rules (and thus limiting it to SMODELS and PLATYPUS) was used. The test was to verify the equivalence of:

```
! 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 because, although they are clearly equivalent, the modelling and reasoning required to show this is non-trivial. Programs `v8` to `v24` are variants with word lengths of 8 to 24 bits respectively.

The final test suite for *verify* uses two instruction sequences that differed only on one input, thus creating a program with a single answer set. The size of the search space was altered by simply fixing the value of some of the bits (programs `w8` to `w24` test for equivalence over 8 to 24 bits). The instruction sequences were:

```
! input in %i0                ! input in %i0
sra %i1 31 %l1                addcc   %i0 %i0 %l1
orcc %l1 %l1 %o1              subxcc  %i0 %l1 %l2
bne 2                         addx    %l2 %i0 %o1
add %l1 1 %o1                 ! output in %o1
! output in %o1
```

The programs used in these tests are available online[4] and will be contributed to the Asparagus benchmark collection [30].

5.3 Results

Figure 3 gives the number of atoms, answer sets and rules, along with the sizes of the search spaces, for the programs used in the tests. Timing results for the *search* tests are given in Figure 4, while the results for the two *verify* tests are given in Figures 5 and 7 and graphed in Figures 6 and 8 respectively. Not all tests could be completed due to time constraints, with entries marked $> n$ aborted after n seconds and those marked $-$ not attempted due to estimated compute times. Results for PLATYPUS on a single node are not presented, due to limitations in the current MPI implementation.

5.4 Analysis

The results presented suggest a number of interesting conclusions. Firstly, answer set solvers are capable of handling non-trivial, real-world problems and attempting to solve the problem of generating optimal code with them is viable. Although the compute time for the space of four instructions is prohibitively high, it is worth noting that this is without even obvious constraints (such as the output of each instruction apart from the

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

[4] `http://www.bath.ac.uk/~mjb/toast/`

Program	Atoms	Rules	Raw search space	Program	Atoms	Rules	Raw search space
v8	975	1755	2^8	w8	2296	12892	2^8
v9	1099	2063	2^9	w9	2296	12894	2^9
v10	1235	2402	2^{10}	w10	2296	12896	2^{10}
v11	1383	2772	2^{11}	w11	2296	12898	2^{11}
v12	1543	3173	2^{12}	w12	2296	12900	2^{12}
v13	1715	3605	2^{13}	w13	2296	12902	2^{13}
v14	1899	4068	2^{14}	w14	2296	12904	2^{14}
v15	2095	4562	2^{15}	w15	2296	12906	2^{15}
v16	2303	5087	2^{16}	w16	2296	12908	2^{16}
v17	2527	5645	2^{17}	w17	2296	12910	2^{17}
v18	2763	6234	2^{18}	w18	2296	12912	2^{18}
v19	3011	6854	2^{19}	w19	2296	12914	2^{19}
v20	3271	7505	2^{20}	w20	2296	12916	2^{20}
v21	3543	8187	2^{21}	w21	2296	12918	2^{21}
v22	3827	8900	2^{22}	w22	2296	12920	2^{22}
v23	4123	9644	2^{23}	w23	2296	12922	2^{23}
v24	4431	10419	2^{24}	w24	2296	12924	2^{24}
s1	6873	1197185	129				
s2	6873	1197184	35862				
s3	6873	1197183	15241350				
s4	6873	1197182	9190534050				

Fig. 3. Program Sizes

Program	s	u	m	$p/2$	$p/4$	$p/8$	$p/16$
s1	42.93	1736.11	60.95	140.327	141.824	139.167	141.167
s2	214.31	66357.55	235.08	256.68	397.393	410.122	461.69
s3	74777.67	$>$ 304401.98	$>$ 407556.46	51580	19523.7	9758.36	7503.68
s4	$>$ 237337.35	-	-	-	-	-	-

Fig. 4. Search Test Times (secs)

last must be used, no instruction or argument pair should be repeated, etc), let alone some of the more sophisticated constraints (such as removing all non-optimal pairs and triples). Thus implementing *search* using ASP seems eminently possible. The results for *verify* are less encouraging and suggest that attempting to verify sequences using greater than 32 bits of input is likely to require significant resources given current solver technology.

The results also suggest a number of interesting points related to solver design and implementation. Firstly, clearly implementation does matter. SURYA implements a slight refinement of the algorithm used in SMODELS, but performs significantly worse in almost all cases. How serious these implementation differences are is not known, but clearly for any solver that is intended to be competitive, implementation details do matter. Another, more subtle issue suggested by these results is the cost of *lookahead*. In the first verify test, the times increase significantly faster that doubling, despite the search space itself only doubling. In the second test, the rate of increase is much closer to doubling. In the first case, the increasing number of atoms, and thus the rising cost of *lookahead* is thought to cause this disparity. This fits with other experiments that have been run using the TOAST programs and explains why NOMORE++ is gener-

Program	s	$p/2$	$p/4$	$p/8$	$p/16$
v8	0.153	0.497732	0.560709	0.633932	0.721136
v9	0.306	0.866704	0.70772	0.808055	0.935053
v10	0.675	1.61512	1.2337	1.16333	1.39326
v11	1.537	3.42153	1.97181	1.93191	2.2948
v12	3.597	7.46042	4.28284	3.53243	3.38788
v13	8.505	15.8174	8.86814	6.25371	6.22179
v14	17.795	34.4004	19.5743	12.38	9.52772
v15	39.651	77.0911	41.1235	27.2365	15.3818
v16	93.141	167.222	71.3785	46.6144	35.3159
v17	217.162	372.57	146.603	99.3623	72.3708
v18	463.025	815.373	384.237	189.690	122.038
v19	1002.696	1738.02	681.673	421.681	262.611
v20	2146.941	3790.84	1514.80	896.705	566.040
v21	4826.837	8206.4	3438.71	1874.36	1244.95
v22	11168.818	17974.8	6683.06	3850.71	2296.87
v23	23547.807	38870.5	15047	7947.95	4833.66
v24	52681.498	83405.1	32561.2	16165.4	10580.4

Fig. 5. First Verify Test (secs)

ally slower than SMODELS. Interestingly, the second verify test also has NOMORE++'s times increasing by less than a factor of two as the search space doubles, suggesting that, although more costly, its branching heuristic is indeed 'smarter'. Again this fits with other tests, which have found degenerate *verify* programs where NOMORE++'s branching heuristic has performed significantly better than any other solver.

Finally, the results suggest some interesting possibilities in the field of distributed solver development. The performance of PLATYPUS on s3 and v16 to v24 demonstrates the power of the technique and that, especially for larger programs, near-linear speed up is possible. However, the performance on s1, s2 and v8 to v15 also shows that, unsurprisingly, on smaller programs the overhead of distribution outweighs the benefits. Why PLATYPUS takes longer than SMODELS on w8 to w24 is not known. Potentially, the smaller number of atoms meant the program's balance between *expand*, *lookahead* and branching were not of the right form to demonstrate the value of distribution or problems with the delegation policy. Parallel solvers are clearly a very powerful advance in solver technology, but one that must be used carefully.

6 Future Development

One of the key targets in the next stage of TOAST development 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 improving 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. A slight modification to *search* allows it 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 classes of instructions. Only the shortest member of each class needs to be in the search space

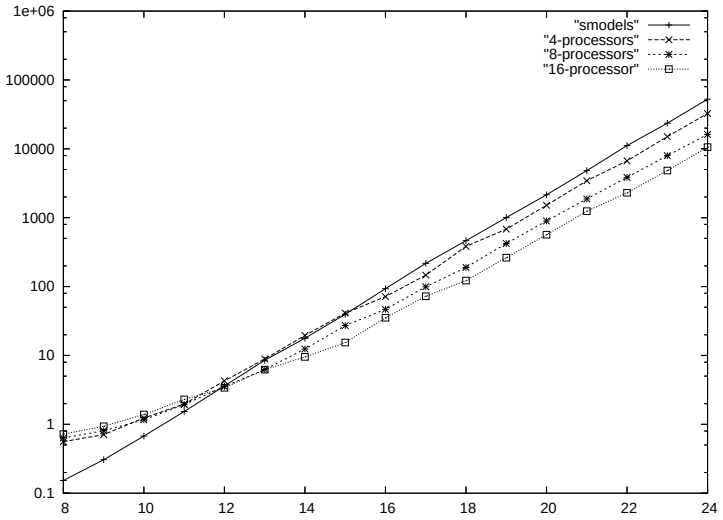

Fig. 6. First Verify Test Timings (log scale)

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 the TOAST system is ported to and will give significant improvements in terms of search times. The equivalence classes generated may also be useful to augment *verify*.

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. In its current form, it is usable for some of the smaller, embedded processors, though 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 ASP program generated, reducing the instructions to a minimal/pseudo-normal form (an approach first used by Massalin), using some form of algebraic theorem proving (as the Denali project used) or attempting to use 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 toolchains such as GCC is also a key target for the project. By implementing tools that translate between the TOAST internal format and processor-specific assembly language, it will be possible to check the output of GCC for sequences that can be superoptimised. Patterns that occur regularly could be added to the instruction generation phase of GCC. The code generators used by JIT (Just In Time) compilers and performance critical system libraries, such as GMP (GNU Multiple Precision Arithmetic Library) could also be 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

Program	s	u	m	$p/2$	$p/4$	$p/8$	$p/16$
w8	0.63	10.939	3.05	3.90092	5.31244	5.24863	5.6869
w9	0.75	20.077	4.78	5.18021	5.65978	7.06414	6.04698
w10	1.02	35.487	8.03	6.67866	6.64935	7.29748	7.87806
w11	1.67	72.561	13.83	10.2767	10.0379	10.7023	9.45804
w12	2.79	129.669	25.67	17.4263	15.9894	15.3256	14.996
w13	5.03	248.974	45.83	32.1642	27.4496	27.4135	22.0735
w14	8.99	541.228	88.23	60.5059	54.06	47.4698	40.7822
w15	18.46	1019.908	161.95	118.506	102.141	77.1823	68.1758
w16	32.55	1854.699	303.69	232.18	189.572	174.96	136.873
w17	69.06	3918.655	554.62	460.882	386.886	357.874	252.538
w18	128.03	7245.888	1034.30	910.266	774.498	653.251	467.091
w19	254.43	14235.360	1898.05	1815.91	1602.51	1311.38	885.655
w20	526.03	27028.049	3576.83	3647.52	3012.45	2558.88	1527.01
w21	1035.09	60064.824	6418.55	7306.26	6061.85	4715.55	3378.42
w22	2552.65	109205.951	11910.02	14813.1	11279.8	9499.05	6788.65
w23	4091.70	238583.922	22766.03	-	23948.4	18912.2	11793.5
w24	8730.61	-	43161.32	-	47673.3	37630.4	23156.8

Fig. 7. Second Verify Test (secs)

for solver performance and the basis of other applications which reason about machine code.

7 Conclusion

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

However, it is not without challenges. Although savings to both the 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 it requires, 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 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 needs to be done to make it competitive.

References

1. I. Niemelä and P. Simons: 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., Springer (1997) 420–429
2. Thomas Eiter and Nicola Leone and Cristinel Mateis and Gerald Pfeifer and Francesco Scarcello: The KR System DLV: Progress Report, Comparisons and Benchmarks. In: Proceedings of the 6th International Conference on the Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann (1998) 406–417

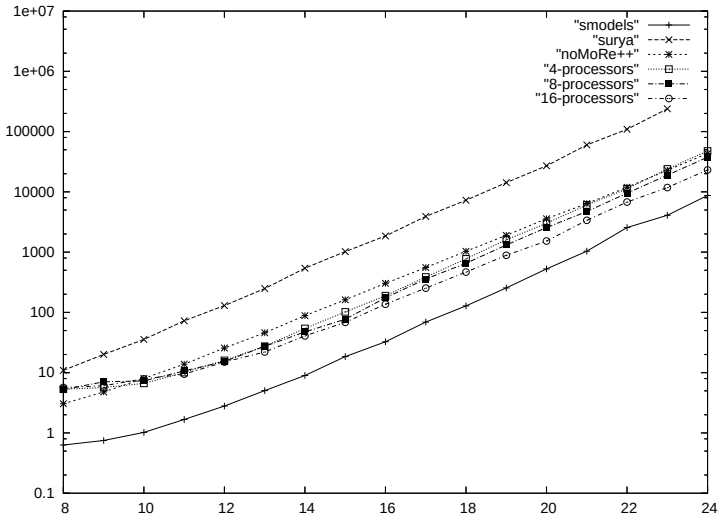

Fig. 8. Second Verify Test Timings (log scale)

3. WASP: WP5 Report: Model Applications and Proofs-of-Concept. `http://www.kr.tuwien.ac.at/projects/WASP/wasp-wp5-web.html` (2004)
4. Thomas Eiter and Wolfgang Faber and Nicola Leone and Gerald Pfeifer and Axel Polleres: Using the DLV System for Planning and Diagnostic Reasoning. In: Proceedings of the 14th Workshop on Logic Programming (WLP'99). (2000) 125–134
5. Monica Nogueira and Marcello Balduccini and Michael Gelfond and Richard Watson and Matthew Barry: 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). Volume 1990 of LNCS., Springer (2001) 169–183
6. De Vos, M., Crick, T., Padget, J., Brain, M., Cliffe, O., Needham, J.: 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., Springer (2006) 72–88
7. P. Giorgini, F. Massacci, J. Mylopoulos and N. Zannone: 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., Springer (2004) 176–190
8. S. Costantini and A. Formisano and E. Omodeo: Mapping Between Domain Models in Answer Set Programming. In: Proceedings of Answer Set Programming: Advances in Theory and Implementation (ASP'03). (2003)
9. Massalin, H.: 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), IEEE Computer Society Press (1987) 122–126
10. Aho, A.V., Sethi, R., Ullmann, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
11. Appel, A.W.: Modern Compiler Implementation in C. Cambridge University Press (2004)

12. Proebsting, T.: Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. http://research.microsoft.com/~toddpro/papers/law.htm (1998)
13. Granlund, T., Kenner, R.: 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), ACM Press (1992) 341–352
14. Joshi, R., Nelson, G., Randall, K.: Denali: A Goal-Directed Superoptimizer. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), ACM Press (2002) 304–314
15. Joshi, R., Nelson, G., Zhou, Y.: The Straight-Line Automatic Programming Problem. Technical Report HPL-2003-236, HP Labs (2003)
16. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Proceedings of the 5th International Conference on Logic Programming (ICLP'88), MIT Press (1988) 1070–1080
17. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9**(3-4) (1991) 365–386
18. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
19. Denecker, M.: What's in a Model? Epistemological Analysis of Logic Programming. In: Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR2004), AAAI Press (2004) 106–113
20. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. to appear in ACM Transactions on Computational Logic (2006)
21. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Communications of the ACM **5**(7) (1962) 394–397
22. Syrjänen, T.: Lparse 1.0 User's Manual. Helsinki University of Technology. (2000)
23. Giunchiglia, E., Lierler, Y., Maratea, M.: SAT-Based Answer Set Programming. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), AAAI Press (2004) 61–66
24. Fangzhen Lin and Yuting Zhao: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
25. Enrico Pontelli and Marcello Balduccini and F. Bermudez: Non-Monotonic Reasoning on Beowulf Platforms. In: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03). Volume 2562 of LNAI., Springer (2003) 37–57
26. Kane, G.: MIPS RISC Architecture. Prentice Hall (1988)
27. SPARC International, Inc: The SPARC Architecture Manual, Version 8. (1992)
28. Mellarkod, V.S.: Optimizing The Computation Of Stable Models Using Merged Rules. Technical report, Texas Tech University (2002)
29. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In: Proceedings of Answer Set Programming: Advances in Theory and Implementation (ASP'05). (2005)
30. Asparagus Project Team: Asparagus Benchmark Project. http://asparagus.cs.uni-potsdam.de/ (2004)