Technical Report

An Application of Answer Set Programming: Superoptimisation (A Preliminary Report)

Martin Brain, Tom Crick, Marina De Vos and John Fitch
An Application of Answer Set Programming: Superoptimisation
A Preliminary Report*

Martin Brain, Tom Crick, Marina De Vos and John Fitch
April 2006

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.

*This paper was presented at and accepted for publication in the informal proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR’06), Lake District, UK.
1 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 [1, 2]. Initial studies have demonstrated [3] that it has great potential in many application 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 for the NASA Space Shuttle [5]. 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. 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 [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 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.

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 solve 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 (identification of common sub-expressions and constant folding) [10] to the costly and esoteric (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). 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.

\(^1\)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.
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 [12].
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.

### 2.2 Superoptimisation

Superoptimisation is a radically different approach to code generation, first described in [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 
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 [9], which returns the sign of a binary integer, or zero if the input is zero:

```c
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
subccc %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) [13] 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 [14, 15]. 
Their approach was much closer to that of the TOAST system, using automatic theorem-proving 
technology to handle the large search spaces.

### 2.3 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 in-
structions 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 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

\(^2\)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 efficient code for them.
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 [13].

3 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 [16, 17]. For reasons of
compactness, this paper only includes a brief summary of answer set semantics; a more in-depth
discussion 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 ¬) is knowing that the literal is not true. For example:

\[ a \leftarrow b, \text{not } c. \]

\[ \neg b \leftarrow \text{not } a. \]

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]). 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 ¬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 [1] and DLV [20]) use a DPLL-style algorithm [21].
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 lparses [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 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 [23]. More recent work has
investigated using ‘Beowulf’-style parallel systems to explore possible models in parallel [24]. One
such system, platypus [25], is used in the TOAST system.

4 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 superop-
timisation 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 frag-
ments 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.

4.1 System Components

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

**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.

### 4.2 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.

### 4.3 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 property 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 branches and multiword arithmetic, the flags are a source of many superoptimised sequences and are thus of prime importance when modelling.
Brain, Crick, De Vos and Fitch / An Application of Answer Set Programming: Superoptimisation

\[
\text{value}(C,T,B) \leftarrow \text{istream}(C,P,\text{lxor},R1,R2,\text{none}), \text{pc}(C,P,T), \\
\text{value}(R1,B), \neg \text{value}(R2,B), \\
\text{register}(R1), \text{register}(R2), \text{colour}(C), \\
\text{position}(C,P), \text{time}(C,T), \text{bit}(B).
\]

\[
\text{value}(C,T,B) \leftarrow \text{istream}(C,P,\text{lxor},R1,R2,\text{none}), \text{pc}(C,P,T), \\
\neg \text{value}(C,R1,B), \text{value}(C,R2,B), \\
\text{register}(R1), \text{register}(R2), \text{colour}(C), \\
\text{position}(C,P), \text{time}(C,T), \text{bit}(B).
\]

\[
\neg \text{value}(C,T,B) \leftarrow \text{istream}(C,P,\text{lxor},R1,R2,\text{none}), \text{pc}(C,P,T), \\
\neg \text{value}(C,T,B), \text{register}(R1), \text{register}(R2), \text{colour}(C), \\
\text{position}(C,P), \text{time}(C,T), \text{bit}(B).
\]

\[
\text{symmetricInstruction}(\text{lxor}).
\]

Figure 2: Modelling of a Logical XOR Instruction in ASP

<table>
<thead>
<tr>
<th>Length of Sequence</th>
<th>No. rules</th>
<th>Grounding time</th>
<th>No. ground rules</th>
<th>No. of atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>530</td>
<td>20.100</td>
<td>95938</td>
<td>1018</td>
</tr>
<tr>
<td>2</td>
<td>534</td>
<td>65.740</td>
<td>298312</td>
<td>1993</td>
</tr>
<tr>
<td>3</td>
<td>538</td>
<td>142.22</td>
<td>643070</td>
<td>3428</td>
</tr>
<tr>
<td>4</td>
<td>542</td>
<td>-</td>
<td>1197182</td>
<td>6873</td>
</tr>
</tbody>
</table>

Figure 3: Search Program Sizes

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]). 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 \text{lxor} is symmetric, which is used to reduce the search space.

The input vectors and output constraints are the program fragments created by \text{pickVectors} and \text{execute} respectively.

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

5 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 \text{smodels} v2.28 (denoted \(s\)) and the initial MPI version of \text{platypus} running on \(n\) nodes (denoted \(p/n\)). \text{lparse} v1.0.17 was used in all cases to ground the programs. The timings displayed are from the \text{smodels} internal timing mechanism and the \text{platypus} MPI wall time respectively. Values for \text{lparse} are the user times given via the system \text{time} command. Results for \text{platypus} on a single node are not presented, due to limitations in the current MPI implementation.

5.1 Search Time

\text{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  %i2
add  %i0  %i1  %i3
add  %i0  %i2  %i3
sub  0    %i3  %o1
! output in %o1
```

This sequence was selected as a ‘worst case’, an example of a sequence that cannot be super-optimised, 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.

5.2 Verification Time

\text{verify} was used to create a verification program for the following two code sequences:
using the SPARC-V8\textsuperscript{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.

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

Figure 5: Verification Program Sizes

5.3 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.

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

\textsuperscript{3}SPARC-V8 is a later, minimal extension of SPARC-V7 with the addition of the umult instruction.
### 6 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, it 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) [26] 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.

7 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 has demonstrated that answer set technology is ready to be used in large-scale applications, although more work is required to make it competitive.
References


