Generating Optimal Code using Answer Set Programming

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

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

Abstract This paper presents the *Total Optimisation using Answer Set Technology* (TOAST) system, which can be used to generate optimal code sequences for machine architectures via a technique known as superoptimisation. Answer set programming (ASP) is utilised as the modelling and computational framework for searching over the large, complex search spaces and for proving the functional equivalence of two code sequences. Experimental results are given showing the progress made in solver performance over the previous few years, along with an outline of future developments to the system and applications within compiler toolchains.

1 Introduction

Within the field of compiler development the term *optimisation* is something of a misnomer. Compilers typically use a series of templates to generate machine level instructions from the parse tree of the target program. An optimisation phase [1] then attempts to improve this code (with respect to both size and performance) by applying a set of transforms, reductions and equivalences. In many modern compilers, this results in significant improvements but it is very unlikely to produce optimal sequences of instructions; and if it does, it will not be able to determine that they are indeed optimal. To further complicate matters, it is often not clear which order these improvement techniques should be applied as they may enable or inhibit further improvements. The current order of application in most compilers is a result of experience and trial and error rather than design.

In a relatively narrow, but significant range of applications, this approach to code generation is not sufficient. In the inner loops of high-performance computing tasks, performance critical system libraries, many embedded applications [6] and even the templates used for code generation within compilers [5] (both conventional and Just In Time (JIT) compilers within virtual machines), if it is possible to generate optimal code, then it would be desirable to do so.

Superoptimisation [8] is an approach that views code generation for loop-free segments of code as a combinatorial search problem. Thus by utilising appropriate search techniques it is possible to generate genuinely optimal instruction sequences. The *Total Optimisation using Answer Set Technology* (TOAST) system uses answer set programming (ASP) [3] as a computational framework to solve the superoptimisation search problem. A model of the machine architecture is created in *AnsProlog* and answer set solvers are used to generate and verify candidate optimal instruction sequences. In this way, developments in solver technology can thus directly improve the performance of the superoptimiser. The flexibility of *AnsProlog* also allows arbitrary constraints to be added to the search with minimal effort, something that is very difficult in the case of procedural superoptimisers, but of huge importance, as it allows a superoptimiser to be used to augment its own set of constraints. For reasons of compactness, this paper does not include a description of the answer set semantics or ASP; an in-depth description can be found in [3].

2 Superoptimisation

Massalin [8] coined the term superoptimisation to refer to an alternative approach to code generation for short, loop-free sections of machine code. Rather than starting with crude, template-generated code and running multiple improvement passes, a superoptimiser starts with the specification of a function and performs a directed search for a sequence of instructions that meets this specification.

Superoptimisation naturally decomposes into two tasks: *searching* for candidate sequences that meet a reduced set of conditions and then *verifying* that they meet the required specification. The raw search space of possible sequences of a given length is at least exponential in the number of instructions; potentially factorial if the order of inputs to the sequence is considered. However, a number of constraints and heuristics exist that can considerably reduce the space that has to be searched. For example, if an instruction computes a commutative function (such as addition) then only one ordering of inputs needs to be considered; likewise, if instructions can be reordered then only one ordering need by searched. Handling the size and complexity of this space is the current limit on superoptimiser performance.

Despite significant potential, superoptimisation has received relatively little research within the field of code generation and optimisation. Recent work [5] has utilised a range of techniques to handle the large search spaces involved in superoptimisation, including automatic theorem proving [7] and satisfiability testing [2], showing the viability of the approach for specific application areas. However, a major deficiency of the existing superoptimising implementations is that there is no guarantee of optimality. Due to the significant computational burden of proving the functional equivalence of two non-trivial sequences of code, most of the existing implementations use a representative test to shortcut the verification, or timeout and discard sequences that take too long to verify.

3 The TOAST System

The TOAST provably optimal code generation system consists of modular interacting components that generate answer set programs and parse answer sets, with a controlling interface that utilises these components to generate a shorter, superoptimised version of the original input sequence. A preliminary version of the TOAST system was first presented in [4].

```
in: v32
in: v32
inst: land i1 i2
inst: add i1 1
inst: add i1 2
inst: sub i0 3
out: v32
```

Listing 1. A program in TOAST input format

3.1 Architecture

The TOAST system supports multiple processor types, with processor specific information stored in a description file which provides meta-information about the processor, as well as which instructions are available. The TOAST system currently supports the following architectures: MIPS R2000, SPARC V7 and SPARC V8, with more proposed. 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.

TOAST accepts programs in an assembly language-like format as input. These are used as the target of the search, to find the shortest sequence of instructions that has the same output. The example given in Listing 1 defines a program of four instructions, with two 32 bit inputs and one 32 bit output.

We assume that the cost of each instruction in the input program is the same; for RISC-like processors where there are no cache or memory issues, and no pipeline breaks this is a fair simplifying assumption. In the case of minimising memory taken by the instruction stream, as might be used in mobile devices, this is the correct measure.

A set of vectors, binary values for each bit of each input, are generated in ASP. This give a set of vectors for each possible path through the input code. The input program is then ‘run’ with these vectors to generate constraints, giving the ‘correct’ values of the outputs for each set of vectors.

By using the input vectors and output constraints (essentially start and end values), we search for candidate sequences of length one, two, and so on, up to one less than the length of the input sequence. The set of instruction sequences given by this necessarily contains any optimal sequences, but may contain extra sequences that only give the correct output for one particular set of inputs. Thus once a candidate set has been found, TOAST searches within this candidate set, picking new vectors each time, until either the the set of candidates is empty, in which case the search moves on to the next length, or until the set of candidates stabilises.

When one or more candidate is found, they have to be verified for equivalence to the original sequence, over all inputs. Searching can generate a large amount of candidate sequences, so two verification steps are performed: an initial representative heuristic test and a full equivalence test. *pre-verify* is a fast heuristic that uses a directed set of vectors to perform a fast representative verify on the two programs. If *pre-verify* returns false, then the candidate is discarded (i.e. it is definitely not equivalent); if true, then a full verify must be performed to prove full equivalence. Empirical evidence suggests that the *pre-verify* heuristic is important in quickly discarding invalid sequences, but it is still necessary to validate a sequence with a full verify, as it is possible to generate cases which can pass the heuristic, but will fail the full verification.

```
value(C,T,B) :- istream(C,P,land,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,land,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(land).
```

Listing 2. Logical AND (`land`) instruction encoded in *AnsProlog*

As noted in Section 2, the number of combinations covered by search is factorial in the length of the sequences. In practise this is handled by a series of calls to a solver with progressively increasing program sizes. Likewise verify searches for a refutation in a space of combinations that is exponential in the number of input bits; effectively a co-NP task and handled by a single invocation of a solver.

One key recent development is the *buildMultiple* tool which uses TOAST to build and refine a series of additional constraints which augment the search component. It is based on the observation that an optimal sequence of instructions will not contain a sub-optimal instruction sequence. The search component of TOAST is used to generate a set of all possible instructions sequences of a given length using a fixed number of inputs. These are then superoptimised using TOAST; if they are sub-optimal or equivalent to another sequence then they are abstracted to form additional constraints. Although this procedure is time consuming, it produces very strong sets of constraints and only ever needs to be run once for a given architecture. Critically, it shows a key advantage of using ASP; the flexibility to add extra constraints without changing the search algorithm. With a procedural system, *buildMultiple* would simply not be possible.

3.2 *AnsProlog* Encodings

TOAST uses *AnsProlog* to model the integer processing unit of the target processors. The majority of the model is at bit level, with *AnsProlog* rules relating input bits of an instruction to the output bits.

The *instruction sequence* is represented as a series of facts, or in the case of searching, a set of choice rules. 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 AND (`land`) is given in Listing 2. Note the use of negation as failure to reduce the number of rules needed and the declaration that AND is symmetric, which is used to reduce the search space.

Flow control rules define which instruction will be 'executed' at a given time step by controlling the program counter (`pc`) literal. 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 the abstract property '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. An example encoding is shown in Listing 3.

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

Listing 3. Flow control rules encoded in *AnsProlog*

Flag control rules model the setting and checking 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 [8].

4 Benchmarks

Benchmarks for the two main tasks of the TOAST system are given: searching for candidate sequences and then verification of two sequences to show full equivalence for all inputs.

The tests[1] are for the SPARC V8 [9], a 32 bit RISC architecture family. All tests were run on quad-core Intel 2.8GHz Xeon E5462 processors with 32GB RAM, running a variant of Scientific Linux. Programs were ground with GRINGO (2.0.0) and tested with the following four solvers: CLASP, SMODELS, SMODELS-IE and SUP; all tools were built in 32 bit mode. None of the *AnsProlog* programs generated within the TOAST system require disjunction, aggregates or any other non-syntactic extensions. All programs generated by the TOAST system are tight.

The search test (*sequence4*) attempts to find shorter optimal sequences for a four instruction program, with two 32 bit inputs, as given in Listing 1. This sequence was selected as an example of an optimal sequence that cannot be improved via superoptimisation, giving an approximate ceiling on the performance of the system. Programs *ss1* to *ss4* are searches over the spaces of 1 to 4 instructions respectively.

We performed two types of verification tests: one is which the two programs are (non-trivially) the same, returning zero answer sets (*verifytest1*); and the second in which the two programs differ on only one set of inputs, hence returning one answer set (*verifytest2*). In these tests, we amended the input programs to demonstrate that the TOAST system is able to verify sequences for 8 bit, 16 bit and 32 bit architectures.

Table 1 presents timings for the search and verify tests, with solver time outs occurring after 100 hours. These results demonstrate that we are able to superoptimise sequences for 32 bit architectures, while the projected growth figures suggest that a fully verified build-once architecture library is feasible as is done in *buildMultiple*.

5 Future Work

The results in Table 1 show a significant improvement from the initial benchmarks provided in [4]. Some of this is due to improvements in hardware, although the majority of the improvement is due to the progress made in answer set solvers, particularly the

[1] Available online from: http://www.cs.bath.ac.uk/tom/toast/

	sequence4				*verifytest1*			*verifytest2*		
Solver	**ss1**	**ss2**	**ss3**	**ss4**	**8 bit**	**16 bit**	**32 bit**	**8 bit**	**16 bit**	**32 bit**
clasp-1.1.1	123.20	105.72	578.37	12355.16	0.46	0.48	15.81	0.31	0.37	8.67
smodels-2.32	123.25	266.17	6880.87	-	0.18	11.33	-	0.20	4.75	-
smodels-ie-1.0.0	123.21	281.60	1983.94	-	0.20	11.08	-	0.21	4.79	-
sup-0.2	123.22	103.46	768.36	-	0.40	3.38	-	0.15	0.14	8.70
Atoms	853	1411	2098	2941	904	2212	6940	1030	1526	2518
Rules	42740	118779	238212	410902	1622	4870	17122	3591	6591	12583

Table 1. Timings (in sec) for TOAST search and verify tests for SPARC V8

inclusion of techniques from SAT solvers, notably clause learning. This demonstrates one of the advantages of ASP; that improvements in solver performance directly benefit applications using them, and that more advanced solvers can be 'plugged-in' with minimal integration needed. Our approach incorporates the concept of a full verification rather than a plausibility test as is done in other systems, taking the need for a human out of the operation.

Making use of these advances in solver technology and the flexibility of ASP (especially with *buildMultiple*), it is hoped that TOAST can be built into a competitive superoptimising system. Key application areas are seen in improving the quality of templates and peephole optimisers used in both conventional and JIT compilers.

References

1. Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2nd edition, 2006.
2. Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 394–403. ACM Press, 2006.
3. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
4. Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying Answer Set Programming to Superoptimisation. In *ICLP 2006*, volume 4079 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.
5. Torbjörn Granlund and Richard Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*, pages 341–352. ACM Press, 1992.
6. Mary Hall, David Padua, and Keshav Pingali. Compiler Research: the Next 50 Years. *Communications of the ACM*, 52(2):60–67, 2009.
7. Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A Goal-Directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 304–314. ACM Press, 2002.
8. Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126. IEEE Computer Society Press, 1987.
9. SPARC International, Inc. *The SPARC Architecture Manual, Version 8*, 1992. Revision SAV080SI9308.