University of Bath

PHD

Superoptimisation: provably optimal code generation using answer set programming

Crick, Tom

Award date:
2009

Awarding institution:
University of Bath

Link to publication

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

Copyright of this thesis rests with the author. Access is subject to the above licence, if given. If no licence is specified above, original content in this thesis is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 International (CC BY-NC-ND 4.0) Licence (https://creativecommons.org/licenses/by-nc-nd/4.0/). Any third-party copyright material present remains the property of its respective owner(s) and is licensed under its existing terms.

Take down policy
If you consider content within Bath's Research Portal to be in breach of UK law, please contact: openaccess@bath.ac.uk with the details. Your claim will be investigated and, where appropriate, the item will be removed from public view as soon as possible.

Download date: 14. May. 2025

Superoptimisation: Provably Optimal Code Generation using Answer Set Programming

submitted by

Tom Crick

for the degree of Doctor of Philosophy

of the

University of Bath

August 2009

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author . . . . . .

Tom Crick

Acknowledgements

I would first like to thank my family for their unwavering moral (and occasional financial) support and understanding throughout my time at Bath. But especially to my girlfriend Sara, who has had to put up with someone who has been permanently attached to his laptop for the past two years and has not shared his evenings and weekends – thank you.

From an academic perspective: to Marina De Vos and John Fitch, my supervisors, without whose guidance, support and understanding, I would not have been able to complete this thesis. It has been a long journey, but ultimately a fruitful one. I certainly hope to continue collaborating with both Marina and John in future projects.

Dave Donaghy was superb in the months leading up to submission, with great insight into the big picture of my thesis, along with motivation, support and a bit of banter.

I have also been lucky to have shared a research lab with a number of people who have made my time as a PhD student a fun and interesting one: Martin Brain, Owen Cliffe, Tristan Caulfield, Matthew Collinson, Yi-Zhe Song, Manu Tanguy, Adam Dziedzic, Sirapat Boonkrong, Mark Wood, Jan Drugowitsch, Ana Martins and Dalia Khader to name a few.

I would also like to say thank you to Des Watson and Russell Bradford for an enjoyable viva; I imagine everyone who passes must ultimately find their viva enjoyable, but I can easily say that mine consisted of a thoroughly stimulating and interesting discussion of both my work and the wider research domain.

Abstract

Code optimisation in modern compilers is an accepted misnomer for *performance improvement some of the time*. The code that compilers generate is often significantly improved, but it is unlikely to produce optimal sequences of instructions; and if it does, it will not be possible to determine that they are indeed optimal. None of the existing approaches, or techniques for creating new optimisations, is likely to change this state of play.

Superoptimisation is a radical approach to generating provably optimal code, that performs searches over the space of all possible instructions. Rather than starting with naively generated code and improving it, a superoptimiser starts with the specification of a function and performs a directed search for an optimal sequence of instructions that fulfils this specification.

In this thesis, we present TOAST, the *Total Optimisation using Answer Set Technology* system, a provably optimal code generation system that applies superoptimising techniques to optimise acyclic integer-based code for modern microprocessor architectures. TOAST utilises Answer Set Programming (ASP), a declarative logic programming language, as an expressive modelling and efficient computational framework to solve the optimal code generation problem.

We demonstrate the validity of the approach of superoptimisation using Answer Set Programming by optimising code sequences for two 32-bit RISC architectures, the MIPS R2000 and the SPARC V8. We also present an application of the TOAST system as a peephole optimiser, by generating libraries of equivalence classes of all optimal instruction sequences of a given length for a specific microprocessor architecture. While this is a computationally expensive process, it only ever needs to be performed once per architecture. We also provide significant benchmarks for the performance of state of the art domain solver tools, further demonstrating the applicability of Answer Set Programming in modelling complex real-world problems.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

We should forget about small efficiencies, say about 97% of the time: premature optimisation is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth

1.1 Background

Although the word *optimisation* shares the same etymological root as "optimal", it is rare for an optimisation process in computing to produce a truly optimal system. The optimised system may only be optimal for a very specific situation or environment; also, the path to optimality is not always an obvious or intuitive process.

More specifically, within the field of compilers the term optimisation is something of a misnomer. Compilers typically use a series of templates to generate machine-level instructions from an intermediate representation generated from the input program [2]. An optimisation phase 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 unlikely to produce optimal sequences of instructions; and if it does, it will not be possible to determine that they are indeed optimal [98]. To further complicate matters, it is often not clear in which order these improvement techniques should be applied, as they may inhibit rather than enable further improvements. The current order of application of

these optimising phases and improvement techniques in modern compilers is a result of experience rather than by design [36].

This situation is especially apparent in embedded systems and mobile appliances, where engineers need to derive the optimum hardware-software configuration to achieve the application's real-time demands, while minimising power requirements and retaining flexibility [39, 184]. They need to generate code for multiple system-on-a-chip variants, consisting, for example, of processor core, signal processor and graphics device, from a common code base. They need to derive a custom system-on-a-chip configuration from the program structure. Standard libraries need to be automatically tuned for each custom processor configuration [117]. Therefore, the development of efficient compiler toolchains for these resource-critical platforms is of huge importance.

A comparison between the advances in microprocessor performance (with respect to Moore's Law [135]) and compiler performance led to the assertion commonly referred to as Proebsting's Law [154, 160]. This stated that compiler performance doubles every 18 years, in comparison to every 18 months for microprocessors. The main implication from this comparison was that the cost of research and development on compiler technologies was a wasted effort and that resources would be better spent elsewhere. However, this 4% increase in performance per year [154] is actually a significant improvement and is important for a range of resource-critical environments (agreeing in part with the Knuth quote presented at the start of this chapter). There is a clear justification for continued investment in compiler development and in particular, novel optimisation strategies for these new environments [82, 97, 153].

Superoptimisation [131] is a radical approach to generating provably optimal code sequences, that uses exhaustive search to find the optimal code for a given function. Superoptimisation naturally decomposes into two sub-problems: searching for sequences that meet specific criteria and then verifying which of these candidates are functionally equivalent to the input function. It has previously only been studied in the context of computing small mathematical functions or optimising fragments of performance-critical code [79]. Superoptimisation provides a fresh approach to the optimisation problem, by aiming for optimality from the outset.

The goal of this thesis is to apply superoptimising techniques to create a practical toolchain for provably optimal code generation. This thesis applies Answer Set Programming (ASP), an expressive declarative logic programming paradigm for modelling real-world problems, to a significant new application domain. It is used to model the machine architectures and their instructions, along with providing a powerful and

efficient computational framework. By modelling the superoptimisation problem in ASP, we are able to use existing domain tools, known as solvers, to generate solutions. The expressive semantics of ASP enables clear and concise modelling of the complex constraints of the problem domain.

1.2 Motivation

Building upon the previous section, the motivation for this thesis is as follows:

Structured approach to optimisation: none of the existing approaches, or methods for creating new approaches, to optimising code aim for optimality from the outset. Developing a structured framework for applying superoptimising techniques to generating optimal code for modern machine architectures would be a significant step. These new strategies for code optimisation would have potential application to existing compiler toolchains.

Lack of proven optimality: with the existing code improvement algorithms it is theoretically complex and computationally expensive demonstrating the equivalence of optimised code sequences to the original code.

Emergence of new performance-critical domains: especially embedded environments; the development of new models and metrics of optimality, such as program size, execution speed, low memory usage and low power consumption, is hugely importance. A practical and adaptable optimisation framework would be applicable to a large number of microprocessor architectures and environments.

Modelling of real-world problems using Answer Set Programming: the expressive language and clear semantics of ASP, coupled with a powerful computational framework and the wide availability of open source solver tools. ASP is regarded as the computational embodiment of non-monotonic reasoning and a primary candidate for an effective knowledge representation tool. This makes it a clear candidate for modelling the provably optimal code generation problem.

1.3 Main Contributions

The main contribution of this thesis is validation of the approach of applying superoptimising techniques to generating provably optimal code sequences for modern ma-

chine architectures using Answer Set Programming. Furthermore:

- Development of a practical and adaptable superoptimising code generation system based on ASP technology, with proof of optimality for short code sequences. This system will be benchmarked against existing superoptimising implementations.
- Demonstrating that superoptimisation of code is achievable in the general case and that the technique can be applied to generate provably optimal code sequences for modern machine architectures.
- Benchmarks and observations on the performances of a range of ASP solver tools, notably the performance of the more recent SAT-based and clause learning solvers compared to the traditional backtracking solvers.
- Demonstrating that ASP is an suitable language for reasoning about large-scale, real-world problems. The application of ASP to the code optimisation problem will also contribute to the ASP community and further drive tool development.

1.4 Related Publications

The following list includes all publications by the author which are related to this thesis, in accordance with Regulation 16.5 subsection k*(ii)* of the University of Bath Regulations for Students 2008/2009.

[37] *Generating Optimal Code using Answer Set Programming*
Tom Crick, Martin Brain, Marina De Vos and John Fitch
10th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'09)
September 2009, Potsdam, Germany

[24] *TOAST: Applying Answer Set Programming to Superoptimisation*
Martin Brain, Tom Crick, Marina De Vos and John Fitch
22nd International Conference on Logic Programming (ICLP 2006)
August 2006, Seattle, USA

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

11th International Workshop on Non-Monotonic Reasoning (NMR'06)
May 2006, Lake District, UK

1.5 Structure of the Thesis

The remainder of this thesis is organised into the following chapters:

In Chapter 2 (page 7) we introduce compilers and modern compiler toolchains and outline the state of the art of code optimisation. We also present common methods of improving code, the problems faced by optimising phases in compilers and the future of compiler technology.

In Chapter 3 (page 14) we introduce superoptimisation and how it differs from traditional code improvement techniques. We analyse the complexity of the problem and describe the existing superoptimising implementations, providing benchmarks on illustrative code sequences.

In Chapter 4 (page 24) we introduce the Answer Set Programming (ASP) paradigm and its underlying logical formalism *AnsProlog*. We explain its syntax and semantics, introduce existing domain tools and present some successful implementations of ASP technology.

In Chapter 5 (page 42) we present TOAST, the *Total Optimisation using Answer Set Technology* system, a framework for applying superoptimising techniques to optimising code for modern machine architectures. We explain the rationale for its development using Answer Set Programming, and how we model the architectures and semantics of their instructions with this expressive language. The utility of the TOAST system is demonstrated via benchmark tests for the MIPS R2000, a popular 32-bit RISC architecture. We also provide a number of benchmarks tests of the performance of ASP tools applied to TOAST problems.

In Chapter 6 (page 64) we apply the TOAST system to superoptimising code sequences for the SPARC V8 architecture, a popular 32-bit RISC server architecture. We present benchmarks for searching for and verifying optimal sequences on the SPARC V8, along with a discussion on the future application of the TOAST framework.

In Chapter 7 (page 74) we introduce a significant application for the TOAST system as a peephole superoptimiser. We describe the rationale and motivation for ap-

plying TOAST to this application area, in which we generate equivalence classes of optimal sequences for a given instruction length and number of inputs. This library of optimal sequences can then be applied to optimising code in compiler toolchains and other optimisation frameworks.

In Chapter 8 (page 85) we summarise the contributions of this thesis, demonstrating the validity of the TOAST approach to superoptimising code sequences using Answer Set Programming, and discuss future research directions.

Chapter 2

Compilers and Optimisation

optimise
verb. *make the best or most effective use of (a situation or resource)*

The Oxford English Dictionary

2.1 Introduction

The focus for microprocessor architecture development over the past ten years has been in advances in architecture design: data-level parallelism, instruction-level parallelism, multi-threaded and multi-core. Because of this, there has been a distinct lack of emphasis on the importance of developing compilers and producing efficient code for these increasingly complex architectures. The cost of developing a compiler for a new architecture is dwarfed, by orders of magnitude, by the cost of the development of the architecture [154, 156], but the absence of an efficient compiler can effectively kill an architecture; a prime example of this phenomenon would be the limited success for the original Intel IA-64 Itanium architecture, due to the lack of tools that could generate efficient code for it. For a wide range of resource-critical environments, there is a real need for continued research into compiler technology and in particular, new strategies for optimising code [82].

In this chapter, we introduce compilers, their significant phases and processes and how they optimise and generate code for modern architectures. We discuss a number of common techniques for optimising code in modern compiler toolchains, before discussing future key research areas in compiler technology.

2.2 Compilers

Compilers are software systems that transform programs written in higher-level languages into functionally equivalent programs in object code or machine assembly language for execution [2]. This definition can also be widened to include systems that translate from one higher-level language to another, or from one machine language to another, and so on.

A compiler typically consists of a series of phases that sequentially analyse a program and construct new ones, beginning with a sequence of characters constituting a source program to be compiled and ultimately producing object code to be executed on a machine. Examples of modern compiler toolchains include the GNU Compiler Collection (GCC) [62, 66], LLVM+clang [111, 125], Open64 (formerly Pro64) [123, 144], SUIF Compiler System [83, 175], Intel Compiler Suite [90, 93] and the Sun Java JIT compilers [169].

In a generalised compiler system (see Figure 2-1), there are six typical phases:

lexical analysis of the program presented to it and breaking it into the legal tokens (single atomic units) of the language in which the program is written, for example, keywords, identifiers or symbol names. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognise it [2]. There may also be a language-dependent preprocessing phase which supports macro substitution and conditional compilation.

syntactic analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds an intermediate-level representation, such as a parse tree, according to the rules of a formal grammar which define the language's syntax [2]. The intermediate representation is often analysed and transformed by later phases of the compiler.

semantic analysis where the compiler adds semantic information to the parse tree and builds the symbol table, required to record the identifiers used in the program and their attributes. This phase performs static semantic checks required by the source language, such as type checking (checking for type errors), object binding (associating variable and function references with their definitions) or definite assignment (if there is a requirement for all local variables to be initialised before use) [8]. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase.

program analysis is the gathering of information from the intermediate representation generated during the previous phases. Typical analyses include data flow analysis to build use-define chains (data structures that consist of a use, a variable, and all the definitions of that variable that can reach that use without any other intervening definitions), dependence analysis, alias analysis and pointer analysis [3]. The call graph (a directed graph that represents calling relationships between subroutines) and the control flow graph (another graph representation of all paths that might be traversed through a program during its execution) are also built during the analysis phase [8]. Accurate analysis is a prerequisite for performing code optimisation.

code optimisation and improvement where the intermediate representation is transformed into functionally equivalent but faster (or smaller) forms. These include a number of optimising transforms, reductions and equivalences that range from the relatively simple (such as dead code elimination, constant propagation and inline expansion [2]) to ones requiring significant analysis (for example, register allocation and automatic parallelisation [8]).

code generation is the final transformation of the intermediate representation into the output language of the compiler, usually the native machine assembly language. This involves storage and resource decisions, such as which variables to fit into registers and memory, and the selection and scheduling of machine instructions.

The front end of a compiler consists of the early lexical, syntactic and semantic phases that build the intermediate representation, while the back end is usually associated with the code generation phase. Confusingly, some literature use the term middle end to distinguish the generic analysis and optimisation phases from the machine-dependent code generation phase.

For a more detailed description of the phases of a compiler, see Aho et al. [2].

2.3 Code Optimisation

Code optimisation in a compiler is the process of tuning the output to minimise or maximise some attribute of an executable computer program. Most code is written for optimality with respect to execution speed, but this usually translates into optimising for low memory usage. The growth of the embedded domain has highlighted the need

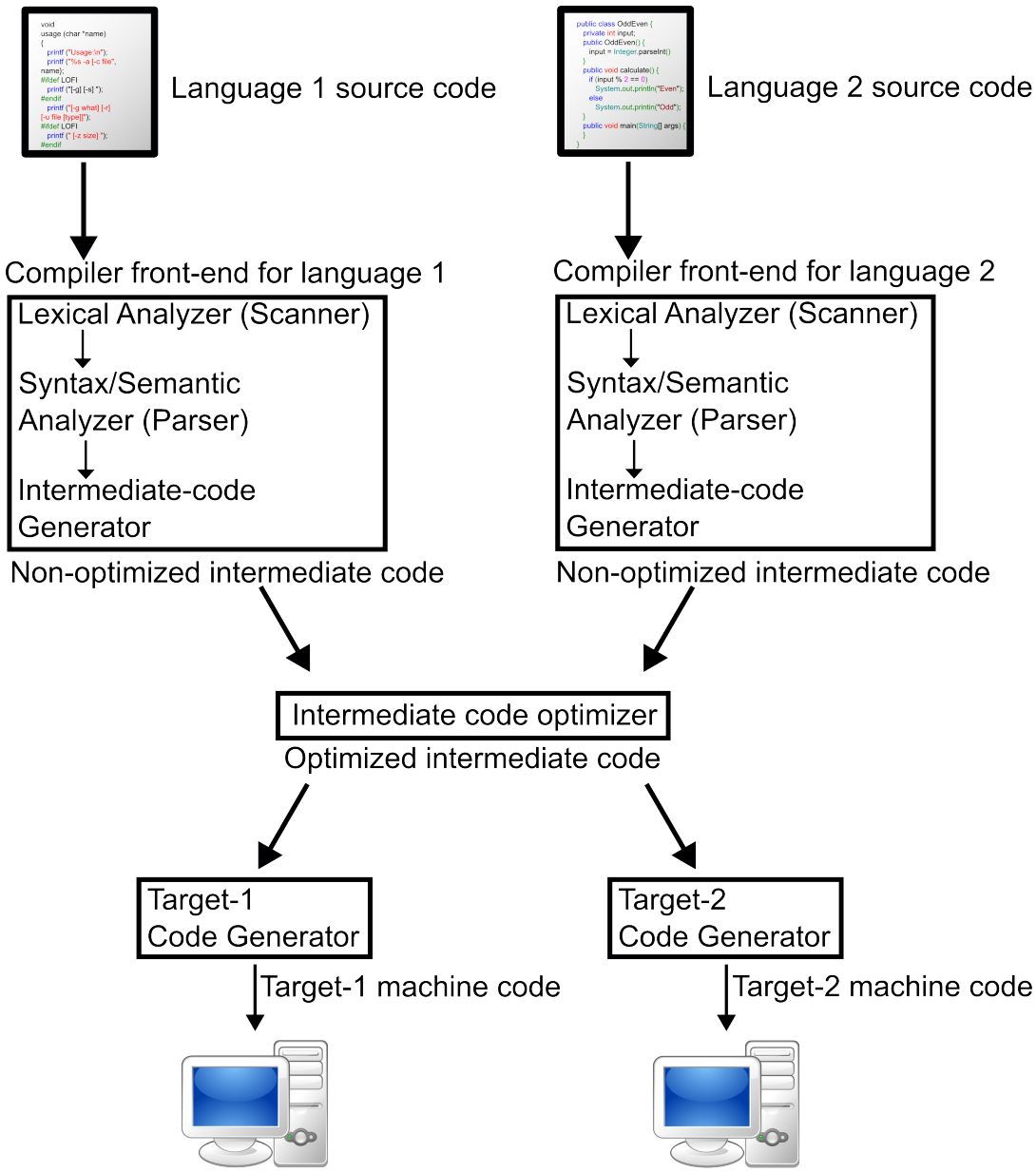

Figure 2-1: Phases of a generalised compiler

for new metrics of optimality, especially minimising the power consumption of a program. The use of numerous optimisation phases to improve the code generated by a compiler can be significant [26, 82].

Techniques used in optimising transformations can be categorised by scope, which can be anything from single statements, to the procedural level, to the entire program. Generally, locally-scoped techniques are easier to implement than global ones (and require less time and fewer resources), but tend to result in smaller gains: an example would be peephole optimisation (described in Section 2.4). Interprocedural or whole pro-

gram analysis [112] can utilise the greater quantity of available information to optimise than when compared to only having access to localised information [2]. However, local optimisations such as loop optimisations, for example loop unrolling and loop fusion [3], act on the statements which make up a loop and can perform loop-invariant code motion. Loop optimisations can provide significant runtime improvements to programs, because many programs spend a large percentage of their time executing code inside loops. It is also common to optimise the object code after the code generation phase [41, 127, 163]. There also exists a range of programming language-specific, machine-dependent and independent optimisations that can significantly improve the generated code [2, 159]. Effective and efficient code optimisation is hard [113], with even designers of modern microprocessor architecture having to write optimisation manuals for developers to effectively use their platforms [4, 5, 88, 91, 92].

For a more detailed overview and discussion of common compiler optimisations, along with the analyses that support them, see [2, 3, 137].

2.4 Peephole Optimisation

A peephole optimisation is a simple but effective technique for locally improving code [133]. It is performed by examining a sliding window of target instructions (the "peephole") and replacing instruction sequences within this peephole by better sequences (depending on the metric of optimality desired). Peephole optimisation can be applied directly after intermediate code generation to improve the intermediate representation [42, 172] or as a post-pass optimisation phase to object code [41, 43].

It is characteristic of peephole optimisation that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to obtain the maximum benefit. Examples of common peephole optimisations include redundant instruction elimination, flow of control optimisations (such as removing jumps and straightlining code), algebraic simplifications (such as strength reduction, where it is less expensive on certain machine architectures to multiply rather than calculating powers, or division by a power of two via shifting), and use of machine-specific features such as auto-increment or auto-decrement addressing modes.

Peepholing techniques exist in some form in most modern compiler toolchains and has seen widespread use in Just In Time (JIT) compilers to dynamically optimise interpreted code at runtime [11, 169].

2.5 Domain Complexity

The work on the complexity analysis of compiler optimisations has proved that certain aspects are NP-complete (or NP-hard), including register allocation [77], instruction scheduling [20, 182] and code generation [65]. In the worst-case, certain features of optimisation have been shown to be exponential or even factorial [107]. Nevertheless, in the general case, optimising and generating code is a tractable problem and significant results are possible.

2.6 The Future of Compiler Technology

Even with the existing levels of code optimisation in modern compiler toolchains, there is a real need for continued research into compiler technology and in particular, in optimising code for new metrics of optimality and new environments, such as multi-core architectures [82]. The emergence of performance-critical environments, such as the embedded domain, with significant low power and low memory consumption requirements [100], means that new metrics of optimality need to be considered and provided; hence optimising for power and memory consumption are important targets for today's compilers [26, 39, 184]. Recent developments in the GCC toolchain has resulted in the implementation of a polyhedral optimisation framework [151, 152, 176], known as GRAPHITE, for high-level memory optimisations. This has significantly improved code generation in GCC, along with improving the selection of loops transforms and adding support for auto-vectorisation [62, 66]. Similar work on speculative analysis and optimisation has also been done for the Open64 compiler [123].

However, the development and implementation of new techniques, such as link-time optimisation and interprocedural optimisation in GCC [29, 112, 174], automatic generation of optimisations using equivalence proofs [173], translation validation of optimisers [98], along with research and development projects such as MILEPOST GCC [63] (which uses machine-learning based self-tuning compilers [146, 168]), ACOVEA [110] (application of genetic algorithms) and the Collective Tuning (*cTuning*) project [64] (developing iterative feedback-directed compilation techniques, collective optimisation, run-time adaptation, statistical analysis and machine learning) are pushing the performance boundaries of modern compilers and optimisation algorithms. Nevertheless, one of the key problems of optimising and generating code for increasingly complex modern archiectures is analysing how the different optimisation techniques interact with each other, and whether they create constrained optimisation problems [183]

when attempting to optimise for more than one metric.

2.7 Summary

This chapter has presented an overview of modern compilers and code optimisation techniques, along with discussing the future of compiler technology. It has been suggested that while the performance improvements of compilers per year is small (in comparison to the improvement in the performance of microprocessors), the increasing demands of today's resource-critical environments require a more structured approach to developing compiler technologies and more importantly, frameworks for supporting more efficient code optimisation.

In the next chapter we present one such approach to code optimisation, which aims to generate truly optimal code sequences from the outset: superoptimisation.

Chapter 3

Superoptimisation

In almost every computation a great variety of arrangements for the succession of the processes is possible...one essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Ada Lovelace

3.1 Introduction

As suggested in the previous chapter, there is a real need for continued research and development in compiler technology, and in particular new strategies to optimising code for different environments. State of the art code optimisation techniques are increasingly specific and rely on complex mathematical specifications; while significant improvements are possible, these techniques are unlikely to produce optimal code, and if they do, it is not possible to determine whether the code produced is indeed optimal.

In this chapter, we introduce superoptimisation, a method for finding the shortest sequence to compute a function. Superoptimisation is an approach that aims for optimality at the outset, distinguishing it from existing code improvement techniques. We describe the motivation for superoptimisation as a technique for generating optimal code sequences, introduce and evaluate existing superoptimising implementations and

provide some example sequences and how they are superoptimised.

3.2 Motivation

Since the earliest days of computer programming, there have been known to exist sequences of machine instructions with seemingly magical properties [1, 6, 18, 19, 138]. By using particular sequences of instructions, often with no obvious connection between them, it is possible to calculate certain functions using fewer instructions than any obvious approach, either hand-coded or compiled from a high-level language. These sequences are inadvertent by-products of the design of the instruction set architecture and, although an entertaining curiosity, seemed not to have an extensive practical use beyond hand-coded assembly language in specific environments, such as in digital signal processing.

However, early work in this area converted sequences of instructions into single instructions [101], based on template matching and utilised as a form of peephole optimisation. Nevertheless, the idea that instruction sets exhibited non-obvious functionality was implicitly understood but never really exploited. Hence, early approaches to generating optimal code [163] by applying this knowledge did not receive much interest in the compiler community.

3.3 The Superoptimisation Problem

Superoptimisation is a radically different approach to code optimisation, first described by Massalin [131]. As noted in later work [95], the term superoptimisation is itself an oxymoron: if a program has been optimised – meaning it is optimal – then what does it mean to superoptimise it? The terminology problem lies in the need to distinguish superoptimisation from garden-variety code improvement techniques. As discussed previously, modern optimising compilers apply a defined set of transforms, reductions and equivalences. This often results in significant code improvement, but it is not necessarily going to produce optimal sequences of instructions; and if it does, it will not be able to determine that they are indeed optimal.

The classical meaning of superoptimisation is to find an optimal code sequence for a loop-free assembly language sequence of instructions. This requirement for straight-line loop-free code was due to the inherent problems of optimising away loops and branches without the appropriate analysis.

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

Listing 3.1: Example C function to calculate the sign of an integer (`signum`)

Rather than starting with naively generated code and improving it, a superoptimiser starts with the specification of a function and performs a directed search for an optimal sequence of instructions that fulfils this specification. Superoptimisation therefore naturally decomposes into two sub-problems: searching for sequences that meet specific criteria and then verifying which of these candidate sequences are equivalent to the original function. As the length of the original sequence increases, the search space increases at a worst-case exponential rate [131]. This has previously made the technique unsuitable for use in standard compiler toolchains, though for improving the code generators of compilers and for targeting key sections of performance critical functions, the results can be significant [79, 96].

A good example of how superoptimisation works is demonstrated by the `signum` function given in Listing 3.1, originally presented by Massalin [131], which returns the sign of an integer, or zero if the input is zero. As you can see from the results presented in Table 3.1 (generated by compiling the function on a Sun UltraSPARC IIIi running Solaris 10, using GCC 3.4.3 and Sun C Compiler 5.8), a basic compilation of this function would generate twelve or more instructions, including at least two conditional branch instructions. An experienced assembly language programmer might be able to implement it in six instructions, with only one conditional branch. Modern compilers can normally achieve similar results.

GNU C Compiler (3.4.3)			Sun C Compiler (5.8)		
Flags	Instructions	Branches	Flags	Instructions	Branches
None	19	4	None	12	4
`-O1`	7	1	`-xO1`	20	4
`-O2`	6	1	`-xO2`	8	2
`-O3`	6	1	`-xO3`	8	2

Table 3.1: Effects of compiler optimisation levels for compiling the `signum` function on SPARC V7

However, superoptimising this function (in this case for the SPARC V7 [12], a 32-bit RISC architecture) produces the sequence of three instructions presented in Listing 3.2. Not only is this sequence shorter in instruction length, it does not require any condi-

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

Listing 3.2: Superoptimised `signum` sequence generated by GSO for SPARC V7

tional branches, a significant saving on modern pipelined processors. This example shows another interesting property of code produced by superoptimisation: it is not immediately obvious how it provides the required functionality, in this case how it computes the sign of a number. The pattern of addition and subtraction essentially 'cancels out', with the actual computation being done by how the carry flag is set and used by each instruction (on the SPARC V7, instructions whose names include `cc` set the carry flag, while `x` denotes instructions that use the carry flag [12]).

3.4 Implementations

3.4.1 Massalin's Superoptimiser

Massalin's original superoptimiser [131] accepts as input a sequence of assembly language instructions which computes a function and then attempts to generate the shortest program which computes that same function. This is done by an exhaustive search over all possible sequences. The search space is defined by a subset of the architecture's instruction set, generating all combinations of these instructions, first length one, then of length two and so on. Each of the generated programs are tested and if found to match the input program in terms of its function, it is returned as a match.

This exhaustive search approach grows exponentially with the number of instructions in the input function, so techniques and heuristics to prune the search space are required. A fast probabilistic test for determining the equivalence of two programs is used, in which a directed set of test input values are used to discard invalid sequences. Also, clearly non-optimal (i.e. impossible) sequences are discarded, as they do not exist in any optimal program. Examples of this include sequences that have the same effect on the machine state as a sequence of shorter length, or sequences that perform operations that destroys the output of a previous instruction. However, none of these techniques compromise the completeness of the search.

The rationale behind this probabilistic test is that in general, the majority of incorrect

candidate programs will fail this simple test, meaning that you only have to perform a full verification test (for all input values) on as few candidates programs as possible. The set of directed test values include edge cases such as byte boundaries and all ones and zeros; if a sequence passes these chosen tests, then all numbers from -1024 to 1024 would be tested. Due to the computational burden of performing a full Boolean test (which would express the function output in terms of Boolean logic operations), it was not deemed feasible to perform full equivalence tests for sequences of more than three instructions in reasonable time [131]. Hence, the probabilistic test was used as a shortcut to validate large numbers of sequences very quickly (c.50000 programs per second), but it was acknowledged that there was a possibility of a sequence passing the probabilistic test but failing a full verification test. Because of this, all sequences were manually checked by hand for correctness.

However, Massalin's superoptimiser was deemed to have limited usefulness as a code generator for a compiler, due to the time taken to find and verify sequences. This was also due to problems with concisely modelling certain features of the instruction set architectures, along with issues of portability: the superoptimiser was written in Motorola 68000 assembly language. Porting to a new architecture would require translation to the assembly language, requiring significant knowledge and experience.

3.4.2 GSO: the GNU Superoptimiser

The GNU Superoptimiser (GSO) [79] is a function sequence generator that applies superoptimising techniques via an exhaustive generate-and-test approach to finding the shortest sequence for a given function. It further developed Massalin's brute force search strategy by attempting to apply constraints whilst generating elements of the search space; so rather than generating all possible sequences and then discarding those that were marked as clearly redundant, it would prune them during generation. One of its main criteria was on the elimination of conditional jumps in sequences by careful modelling of the processor flags, due to the inherent cost of jumps on modern pipelined architectures. However, many of these approaches were architecture-specific.

GSO is a goal-directed superoptimiser, meaning that it is only able to compute optimal sequences for a specific encoded goal function, such as finding the sign of a number (as given in Listing 3.1), rather than allowing any arbitrary sequence of instructions as input to the system. In this way, it is efficient in generating shorter sequences for these goals, but they first have to be efficiently translated and encoded internally before they can be superoptimised. The time complexity of the GSO algorithm is approxim-

ately $\mathrm{O}(mn^{2n})$, where m is the number of available instructions on the architecture and n is the shortest sequence for the goal function. The practical sequence length limit depends on the target architecture and goal function arity; in most cases it is approximately five instructions, but for richer instructions sets it may be lower [79].

GSO was originally developed on the IBM RS/6000 architecture, but was also deployed on a number of other architectures, including the SPARC V7, Motorola 68000, AMD 29K and the Intel 80386. GSO is written in C, rather than machine-specific assembly language, which made porting significantly easier. Its generic structure relied on lookup tables, which mapped instructions to architectures. If you wanted to target a new instruction set or machine architecture, you would need to create new instruction definitions in a special internal format. One of the main successes of GSO was its contributions to optimisation patterns for the GCC toolchain; when it 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 [79].

As with Massalin's superoptimiser, GSO is unable to guarantee that it generates the best possible instruction sequence for all possible goal functions. This is partly due to the fact that only a subset of the instruction set is modelled, but mainly because the generated code sequences are not exhaustively checked. It is therefore possible that the generated sequences are not valid for all input values.

Example 3.1. An example demonstrating how GSO superoptimises the `signum` function introduced in Listing 3.1 is as follows:

- **Length 1:** No sequences are found (5 sec to search)
- **Length 2:** No sequences are found (5 sec to search)
- **Length 3:** 13 sequences generated (10 sec to search)

As you can see from the 13 results presented in Listing 3.3, the output is in a internal GSO format, similar to C, which describes the abstracted instructions for the machine architecture (in this example, the SPARC V7 [12]). This superoptimisation is a demonstration of the speed of GSO in finding small sequences for certain functions quickly, but this does not scale for larger sequences [79].

```
1: r1:=add_co(r0,r0)
       r2:=sub_cio(r0,r1)
       r3:=add_cio(r2,r0)
2: r1:=add_co(r0,r0)
       r2:=sub_cio(r0,r1)
       r3:=add_ci(r2,r0)
3: r1:=add_co(r0,-1)
       r2:=arith_shift_right(r0,0x1f)
       r3:=add_cio(r2,r2)
4: r1:=add_co(r0,-1)
       r2:=arith_shift_right(r0,0x1f)
       r3:=add_ci(r2,r2)
5: r1:=sub_co(0,r0)
       r2:=arith_shift_right(r0,0x1f)
       r3:=add_cio(r2,r2)
6: r1:=sub_co(0,r0)
       r2:=arith_shift_right(r0,0x1f)
       r3:=add_ci(r2,r2)
7: r1:=arith_shift_right(r0,0x1f)
       r2:=sub_co(r1,r0)
       r3:=add_cio(r2,r0)
8: r1:=arith_shift_right(r0,0x1f)
       r2:=sub_co(r1,r0)
       r3:=add_ci(r2,r0)
9: r1:=arith_shift_right(r0,0x1f)
       r2:=sub_co(r1,r0)
       r3:=add_cio(r1,0)
10: r1:=arith_shift_right(r0,0x1f)
       r2:=add_co(r0,-1)
       r3:=add_cio(r1,r1)
11: r1:=arith_shift_right(r0,0x1f)
       r2:=add_co(r0,-1)
       r3:=add_ci(r1,r1)
12: r1:=arith_shift_right(r0,0x1f)
       r2:=sub_co(0,r0)
       r3:=add_cio(r1,r1)
13: r1:=arith_shift_right(r0,0x1f)
       r2:=sub_co(0,r0)
       r3:=add_ci(r1,r1)
```

Listing 3.3: Superoptimised output generated by GSO for `signum` sequence on SPARC V7

3.4.3 Denali Project

The Denali project [95, 96] applies superoptimising techniques to generating optimal code sequences by using automatic theorem proving technology as an intelligent approach to handling the large search spaces. It accepts input in a low-level machine model, similar to C or assembly language, which contains higher-level language constructs. By converting sequences into a form that a matcher tool can accept, axioms are added and directed graphs with equivalence relations on nodes are constructed. If two terms are semantically equivalent, then a relationship would be created between these nodes. By adding the information from the architecture descriptions, a constraint generator reduces the problem to the Boolean satisfiability problem to be solved by a domain tool known as a SAT solver. The output of the solver is a confirmation of the equivalence of the sequence or not.

Preliminary experimental results were presented [95], with the Denali system able to produce minimum-cycle optimal code for the DEC Alpha EV6 processor. Its formal approach was very well documented, with proofs for all of the superoptimising algorithms, but the development of the system stalled. Verification of code sequences was also a significant problem, as in previous superoptimising implementations, especially due to the size of some of the Boolean expressions passed to the domain tools to solve, requiring significant compute time [96]. For large complex sequences, the solver tools timed out and sequences were discarded.

3.4.4 Stanford Superoptimiser

The Stanford superoptimiser [14] applies superoptimising techniques to the automatic generation of peephole optimisers. It uses a learning approach to populate its database of optimisations, which were verified using SAT solving techniques, in a similar way to the Denali system. Experimental results demonstrated the utility of this system for the Intel x86 architecture, but as with previous superoptimising implementations, full verification of sequences timed out if they exceeded a time limit, with the potential of having discarded an optimal sequence. This peepholing library framework was also applied to the problem of binary translation [15], with some success. However, the framework did not develop further beyond an experimental system for the Intel x86 architecture.

3.4.5 Other Implementations

Simple superoptimising techniques have been deployed in a number of applications, including the GNU Multiple Precision Arithmetic Library (GMP) [78] and for simple analysis of branch code generation in GCC [158]. Superoptimising implementations have been created for Microchip's PIC family [162] and the Atmel AVR, an 8-bit RISC architecture, but these have all been proof-of-concept developments with limited success. While generating some interesting results, the problems encountered by the existing implementations has hindered wide-scale uptake of this promising technology.

3.5 Summary

In this chapter we have introduced superoptimisation, a technique for generating optimal code sequences, and how it differs from existing approaches to improving code.

We have also given an overview of existing superoptimising implementations and the common problems encountered, specifically:

- The complexities of modelling the machine architecture and the semantics of its instructions
- The significant time taken to search the large space of all possible instruction sequences for modern machine architectures.
- The complexity and large computational resources required to prove that two code sequences are equivalent for all possible input values.
- The problem of proving that a generated code sequence is indeed optimal.

Due to the difficulties of showing 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. With this approach, there is a possibility of discarding potential optimal sequences.

However, there exists a number of techniques briefly mentioned in Chapter 2 (page 12) that may also support efficient searching of the large spaces for candidate optimal sequences. The use of genetic algorithms [110], machine-learning [63, 64] and evolutionary techniques [87], analysing discrete sections of the search space, may provide a method of pruning the large search spaces, possibly using distributed computational techniques.

Nevertheless, superoptimising techniques have application to other areas, such as the analysis of the design of instruction set architectures. For example, it may be feasible to encode a proposed instruction set architecture, generate code sequences for a given function and feedback into the design if a superoptimiser discovers shorter sequences; hence, a form of superoptimisation-directed instruction set design. Overall, superoptimisation was identified as a aid to the assembly language programmer, with the unexpected nature of the sequences raising questions about the design of instruction sets, along with a better understanding of the interrelations between arithmetic and logic instructions. Another useful application would be the generation of tables containing lists of equivalent sequences for use in a conventional peephole optimiser (as discussed in Section 2.4, page 11).

In Chapter 5 (page 42), we present the TOAST system, a framework for applying superoptimising techniques to generating provably optimal code sequences. In the next

chapter, we first introduce Answer Set Programming (ASP), the modelling language and computational framework which underpins the TOAST superoptimising system.

Chapter 4

Answer Set Programming

The main purpose of a programming language is to help the programmer in the practice of his art.

C.A.R. Hoare

4.1 Introduction

Answer Set Programming (ASP) [1] [73,130] is a declarative logic programming paradigm that allows reasoning about real-world problems in the absence of complete information. It is a powerful and intuitive non-monotonic [50] [2] logic programming language for modelling, reasoning and verification tasks.

ASP describes a problem as a logic program, a set of axioms and a goal statement, under the answer set semantics of logic programming [73] in such a way that the models of the program (answer sets) correspond to the solutions of the problem. Therefore, by encoding the description of the problem domain and the description of what constitutes a solution, solving the problem is reduced to computing the answer sets of the program.

With its clear syntax and formal expressive semantics, combined with efficient domain tools known as solvers, ASP provides an excellent basis from which derived models

[1] ASP has also been referred to in the literature as *A-Prolog*, AnsSet-Prolog, Answer Set Prolog and Extended Logic Programming.

[2] A formal logic whose consequence relation is not monotonic; meaning that adding a formula to a theory never produces a reduction of its set of consequences

may be computed. It differs from traditional logic programming in that it represents solutions to a problem by models or answer sets (see Figure 4-1), rather than by answer substitution or querying [16], such as in Prolog.

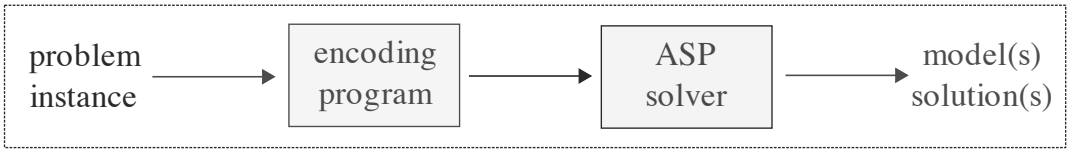

Figure 4-1: Answer Set Programming (ASP) modelling paradigm

In essence, an ASP program can be seen as a formalisation of the underlying reasoning problem in its own right, with the advantage of being able to solve this formalisation directly through the application of solver tools. At this stage, it is important to be clear regarding terminology: Answer Set Programming (ASP) is the programming paradigm, underpinned by the formal language *AnsProlog*.

In this chapter, we describe the history and development of ASP and its roots in non-monotonic reasoning and logic programming. We present its syntax and semantics, along with common extensions. We then give an overview of popular domain tools and successful implementations of ASP technology. We use ASP as a modelling framework throughout this thesis, adhering to the *AnsProlog* notation introduced in Baral [16].

4.2 Origins of ASP

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. Logic is used as a purely declarative representation language, with theorem-proving or model generation as the method of generating solutions to the problem. Logic programming in a narrower sense can be traced back to debates in the late 1960s and early 1970s regarding declarative versus procedural representations of knowledge in artificial intelligence; for example, the PLANNER language for theorem proving in robotics [85].

McCarthy [132] proposed the use of logical formulae as the basis for a knowledge representation language, in which declarative statements were used to express information about the problem. The work in automated theorem proving [106, 126, 157] to prove theorems in first-order logic led to the proposal of the concept of logic programming by Kowalski [104], and later to the first implementation of the programming language

Prolog [33]. The use of Prolog as a practical programming language was given great momentum in 1977 by the development of an efficient compiler [179].

For a comprehensive overview of the history and development of logic programming, see the survey articles [9, 17, 50, 105].

4.3 History of Negation in Logic Programming

The implementation and hence the semantics of negation in logic programming languages is important [9], with different mechanisms for computing when the negation of a predicate is true; a variety of different intuitions of what this means have been proposed [48]. The logical status of negation as failure (i.e. $not(p)$ is true if p cannot be proved using the current information) was unresolved until it was shown that, under certain natural conditions, it is a correct (and sometimes complete) implementation of classical negation [74] (i.e. every proposition is either true or false and cannot be both) with respect to the completion of the program [31]. This is closely related to the closed world assumption [155]: the presumption that what is not currently known to be true, is false. As an alternative to this completion semantics, negation as failure can also be interpreted epistemically, as in the answer set semantics of Answer Set Programming (discussed in more detail later on in this chapter). This means that the epistemic interpretation can be easily combined with classical negation, enabling the formalism of phrases such as *the contrary cannot be shown*, where *contrary* is classical negation and *cannot be shown* is the epistemic interpretation of negation as failure [17].

By linking negation as failure with classical negation, anything that cannot be proven to be true is known to false, essentially assuming that everything that is known about the world is contained in the program. Fages [60] proved that a syntactic condition of logic programs (referred to as *tightness*, which defines the transitive closure of a relation), guarantees the stability of every model of a program's completion. This means that for tight logic programs, the answer set semantics are equivalent to the completion semantics [57]. This idea was further developed by making the completion of a non-tight program stronger by the addition of loop formulae [122] (in which for each loop in the program, a corresponding loop formula is added to the program's completion), so that all the program's non-answer set solutions are eliminated.

From the perspective of knowledge representation, a set of ground atoms (in which all variables have been removed) can be thought of as a description of a complete state of knowledge: the atoms that belong to the set are known to be true and that atoms that do

not belong to the set are known to be false. A possibly incomplete state of knowledge can be described using a consistent, but possibly incomplete set of literals; if an atom p does not belong to the set and its negation does not belong to the set either, then it is not known whether p is true.

Hence, there is a need to distinguish between two types of negation – negation as failure and classical negation. The following example, illustrating the difference between the two types of negation is attributed to McCarthy [74]:

Example 4.1. A school bus may cross the railway tracks under the condition that there is no train approaching. If we do not necessarily know whether a train is approaching, then the rule using negation as failure:

$$cross \leftarrow \mathbf{not}\ train$$

is not an adequate description of this idea: it says that it is acceptable to cross in the absence of information about an approaching train. The weaker rule, that uses classical negation, is preferable:

$$cross \leftarrow \neg train$$

this indicates that it is acceptable to cross if and only if we know there is no train approaching.

4.4 Relationship to Prolog

ASP is a powerful and intuitive non-monotonic logic programming language for modelling, reasoning and verification tasks. One common question asked of researchers working on non-monotonic logic programming systems such as ASP is that *Prolog has been around for many years and is a mature technology, so why not just use that?* The short answer is that Prolog has a number of limitations both in concept and design that make it unsuitable for many knowledge representation and real-world reasoning tasks.

Although Prolog developed out of programming with Horn clauses – a subset of first order logic – several non-declarative features were added to Prolog to make it programmer-friendly, such as the cut operator [35]. The cut is a goal added to a program which cannot be backtracked past, so it prevents extra solutions being generated.

It has been described as a controversial control feature because it is not a Horn clause and was only added for efficiency [124].

The ordering of literals in the body of a rule matters in Prolog as it processes them from left to right. Similarly, the positioning of rules in the program is significant in Prolog, as it processes them from top to bottom. This is not the case for ASP, as a program is a set of *AnsProlog* rules in which the body is a set of literals. Because of this top-down query processing of rules and literals in Prolog, a program may get into an infinite loop for even simple programs without negation as failure.

There are also problems in Prolog dealing with negation as failure: in general, Prolog has trouble with programs that define recursions through the negation as failure operator. *AnsProlog* does not have these problems, and as its name suggests it uses the answer set semantics to characterise negation as failure (as described in Section 4.3, page 26).

Another key difference is that ASP programs have to be explicitly ground to remove variables before being solved; however, given a query, the Prolog engine attempts to find a resolution refutation of the negated query i.e. an instantiation for all free variables is found that makes the union of clauses consisting of the negated query false. It then follows that the original query, with the found instantiation applied, is a logical consequence of the program [34]. However, the method of generating models in ASP is always guaranteed to terminate (though without any constraints on the time taken); this is not the case for Prolog. Even though syntactically ASP programs look like Prolog programs (although Prolog programs do not require explicit grounding) they are treated by rather different computational mechanisms. Indeed, model generation instead of query evaluation can be seen as a recent trend in the encompassing field of knowledge representation and reasoning [50], although there have been efforts to bridge between the two paradigms [53].

4.5 *AnsProlog* Syntax

In this section, we present the *AnsProlog* language that underpins the Answer Set Programming paradigm. For an informal specification of the *AnsProlog* language, see Appendix A (page 110).

A number of syntactic variations of *AnsProlog* exist, the broadest of which is referred to as *AnsProlog**, which denotes there are no restrictions on the rules. Within this thesis, we limit ourselves to the use of a syntactic and functional subset of *AnsProlog**,

referred to as *AnsProlog*$^{\neg,\perp}$, which allows classical negation and the use of constraints.

4.5.1 Core Syntax

Definition 4.1. The *language* of an *AnsProlog* program consists of sets of the following:

1. variables
2. constants
3. n-ary function symbols
4. n-ary predicates
5. connectives
6. punctuation symbols
7. the special symbol $\perp$

Connectives and punctuation symbols are fixed to the sets $\{\ \neg\ \ \leftarrow\ \ \mathbf{not}\ ,\ \}$ and $\{\ (\ )\ .\ \}$ respectively.

In general, variables are expressed as arbitrary sequence of letters that start with an upper-case letter, while constants are expressed as a sequence of characters starting with a lower-case letter. n-ary predicates are expressed as a sequence of characters (the predicate name) starting with a lower-case letter followed by a bracketed list of zero or more arguments. In the case that a predicate has zero arguments, the brackets are omitted. n-ary function symbols are expressed as a sequence of characters starting with a lower-case letter followed by a bracketed list of one or more arguments. To make a clear distinction between the connectives in a first-order theory and the connectives in the languages of an answer set framework, we use different symbols than normally used in first-order theories: *or* instead of $\vee$ and ',' instead of $\wedge$. The above language forms the basis for an *AnsProlog* program in ASP. This language is used to define the terms, atoms and rules which compose a program.

Terms in ASP are recursively defined as follows [16]:

Definition 4.2. A *term* is inductively defined as follow:

1. A variable is a term.
2. A constant is a term.
3. If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

For example, if V is a variable, f is a function symbol with a single argument and c is a constant, then all of the following are terms:

$$c,\ V,\ f(c),\ f(V),\ f(f(c)),\ f(f(f(c)))$$

Terms may be applied to the arguments of predicates to form atoms. Atoms are defined as follows:

Definition 4.3. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and each t_i is a term.

For example, if $p0$ is a predicate with zero arguments, $p1$ is a predicate with one argument and $p2$ is a predicate with two arguments and c, V and $f(c)$ are terms, the following are atoms:

$$p0,\ p1(V),\ p1(c),\ p2(c, V),\ p2(f(c), V)$$

A term is said to be *ground* if no variable occurs in it. An atom is referred to as ground if all of its arguments are ground. Terms and atoms are referred to as unground if they are not ground.

Definition 4.4. A *literal* is either an atom or its classical negation (preceded by the symbol $\neg$). The former is referred to as a positive literal, while the latter is referred to as a negative literal.

A literal is referred to as ground if the atom in it is ground.

An *extended literal* is either an atom or an atom preceded by the symbol $\mathbf{not}$, denoting negation as failure. The former is referred to as a positive extended literal, while the latter is referred as a negative extended literal.

An *AnsProlog*$^{\neg, \perp}$ program is made up of a set of rules.

Definition 4.5. A *rule* is of the form:

$$L_0 \leftarrow L_{k+1}, \ldots,\ L_m,\ \mathbf{not}\ L_{m+1}, \ldots,\ \mathbf{not}\ L_n. \tag{4.1}$$

where L_is are literals or when $k = 0$, L_0 may be the symbol $\perp$, and $k \geq 0$, $m \geq 0$, and $n \geq m$.

The parts on the left and on the right of the '$\leftarrow$' are called the *head* and the *body* of the rule, denoted $Head(r)$ and $Body(r)$ respectively, for a rule r. A rule is said to be ground if all the literals of the rule are ground.

In addition to adding literals to the answer sets of a program, rules can also be used to indicate inconsistencies in a given set of literals. We refer to rules of this form as *constraint rules* (or simply *constraints*). A constraint in *AnsProlog*$^{\neg,\perp}$ is a rule of the form:

$$\perp \leftarrow L_{k+1}, \ldots, L_m, \textbf{ not } L_{m+1}, \ldots, \textbf{ not } L_n.$$

A rule of this form indicates that if the body of the rule is applicable, then the current set of considered literals is not considered as an answer set of the program. In the standard syntax of *AnsProlog*$^{\neg,\perp}$, we may omit the $\perp$ symbol from constraint rules and assume its presence in any rule with an empty head.

We refer to facts in programs as an abbreviation for rules of the form $b \leftarrow \top$, for a given atom b. Within this thesis, we treat the $\top$ as being implicit and assume its presence in any rule with an empty body.

Terms applied to predicates in rules may contain variables, in order to interpret a rule containing variables, the rules is expanded through a process called *grounding*. Grounding translates a program containing variables into a program containing no variables.

In order to ground a program, we must first determine the Herbrand Universe of the program:

Definition 4.6. The *Herbrand Universe* of a language $\mathcal{L}$, denoted by $HU_{\mathcal{L}}$, is the set of all ground terms which can be formed with the constants and function symbols in $\mathcal{L}$.

Definition 4.7. The *Herbrand Base* of a language $\mathcal{L}$, denoted by $HB_{\mathcal{L}}$, is the set of all ground atoms that can be formed with predicates from $\mathcal{L}$ and terms from $HU_{\mathcal{L}}$.

A program Π is grounded by taking each rule in Π and applying each grounded term in $HU_{\mathcal{L}}$ to each variable in each rule of Π. More formally:

Definition 4.8. Let r be a rule in the language $\mathcal{L}$. The grounding of r in $\mathcal{L}$, denoted by *ground*$(r, \mathcal{L})$, is the set of all rules obtained from r by all possible substitutions of elements of $HU_{\mathcal{L}}$ for the variables in r.

For any logic program Π, we define:

$$ground(\Pi, \mathcal{L}) = \bigcup_{r \in \Pi} ground(r, \mathcal{L})$$

and write *ground*(Π) for *ground*$(\Pi, \mathcal{L}(\Pi))$.

The grounding process is best illustrated with an example, using the following unground program:

Example 4.2.

$$\texttt{bird(pigeon)}.$$
$$\texttt{bird(buzzard)}.$$
$$\texttt{has\_feathers(X)} \leftarrow \texttt{bird(X)}.$$

The Herbrand Universe of the program consists of the terms `pigeon` and `buzzard` and these are used to expand the variable `X` in the third rule. The Herbrand Base of this program consists of the atoms `bird(pigeon)`, `bird(buzzard)`, `has_feathers(pigeon)` and `has_feathers(buzzard)`.

The ground version of this program is thus:

$$\texttt{bird(pigeon)}.$$
$$\texttt{bird(buzzard)}.$$
$$\texttt{has\_feathers(pigeon)} \leftarrow \texttt{bird(pigeon)}.$$
$$\texttt{has\_feathers(buzzard)} \leftarrow \texttt{bird(buzzard)}.$$

It should be noted that for an unground program containing function symbols, the Herbrand Universe and Herbrand Base may be infinite. For a ground program, the Herbrand Universe and Herbrand Base will both be finite. Current ASP solver tools only operate on ground programs with finite sets of rules; as a consequence of this, we limit acceptable unground programs to only those which have a finite ground representation. For a further discussion of the case when the Herbrand Universe is infinite, see Baral [16].

In order to constrain programs to those with only finite numbers of rules we introduce two constraints on the structure of rules and programs. The first of these constraints is the range-restriction property, and is specified as follows:

Definition 4.9. An unground rule is *range-restricted*, if each variable in the rule appears in at least one positive atom (not negated by negation as failure) in the body of the rule. A program is range-restricted if all of its rules are range-restricted. The range-restriction property requires that each variable be associated with one or more predicates in the body of the rule.

The second property applies to the whole program and is called the domain-restriction property and is specified as follows:

Definition 4.10. A rule is *domain-restricted* if every variable which appears in the rule also appears in a positive domain predicate in the body of the rule. A program is domain-restricted if all rules of the program are domain-restricted. A predicate is a domain predicate if a ground atom derived from that predicate appears in the head of at least one rule with an empty body (as a fact) and the predicate does not appear in the head of any rules with non-empty bodies.

Example 4.3. For example, the program:

$$\mathtt{p(a)}.$$
$$\mathtt{p(f(X))} \leftarrow \mathtt{p(X)}$$

is range-restricted but not domain-restricted, as the predicate p appears in the head of both a domain-restricted rule and non-domain-restricted rule.

While variables allow a great deal of flexibility and syntactic clarity of programs, it should be noted that in the worst case the number of rules generated by the grounding process may be exponentially larger than the original program.

When speaking about the status of rules with respect to a given set of ground atoms we use the terms *applicable* and *applied*:

Definition 4.11. A rule is said to be applicable with respect to a set of atoms S, if all of the positive literals in the body are in the set: $l_i \in S, 1 \leq i \leq n$, and none of the negated literals are in the set $l_j \notin S, n+1 \leq m$.

A rule is applied if it is applicable and the head atom l_0 is also in the set.

4.5.2 Syntactic Extensions

A number of syntactic extensions to *AnsProlog* have been proposed and are commonly used; the most important of these with respect to this thesis is the use of *choice rules*.

Choice rules are a syntactic extension of *AnsProlog*$^{\neg,\perp}$ for selecting applicable atoms non-deterministically from a set of possible atoms. However, they can be removed from a program by the addition of new rules. A choice rule of the form:

$$h_1, \ldots, h_n \leftarrow L_{k+1}, \ldots, \ L_m, \ \textbf{not}\ L_{m+1}, \ldots, \ \textbf{not}\ L_n.$$

states that if every positive atom in the body of the rule $L_{k+1}, \ldots, \ L_m$ is applied and every negated atom is not applicable, then any subset of $h_1, \ldots, h_n$ is applicable.

Example 4.4. Consider the following program containing the choice rule:

$$a.$$
$$b, c \leftarrow a.$$

The valid answer sets of this program are $\{a\}$, $\{a, b\}$, $\{a, c\}$ and $\{a, b, c\}$.

4.6 Semantics of ASP Programs

Having defined the syntax of *AnsProlog*$^{\neg,\perp}$ programs, we now turn to the intuitive semantics. In this section, we only deal with ground programs (in which variables have been eliminated), where the Herbrand Base of the program is finite.

Given a rule of the form in Definition 4.1, an intuitive definition of the semantics of rules of this form can be as follows: if all positive atoms (those without negation as failure) in the body of the rule: $l_1, \ldots, l_n$ are known to be true and none of the negated literals in the body $l_{(n+1)}, \ldots, l_m$ are known to be true, then the head of the rule l_0 can also be considered to be true. In the case that l_0 is false ($\perp$), then this indicates a contradiction.

An ASP program consists of a set of statements, called rules. Each rule $h \leftarrow B$ is made of two parts, namely the body B, which is a set of extended literals, and a head literal h.

It should be read as: if all of the elements of B are true, so is the head l; or l is

supported if all elements of B are considered to be true. We only assume those literals to be true that are actually supported. This form of reasoning is referred to as the minimal model semantics.

An *interpretation* of a program Π is any set of literals of the program's Herbrand Base: $I \subset HB_\Pi$. We can use interpretations to define *models* of a program, as follows:

Definition 4.12. Let Π be a ground program consisting of rules of the form: $\bot \leftarrow B \in \Pi$ (i.e. constraints) and $l \leftarrow B \in \Pi$, where B is the set of (non-negated) literals in the body of the rule and l is a literal. An interpretation $I \subset HB_\Pi$ of the program Π is a model of the program Π iff for each rule of the program the following is true:

$$h \leftarrow B \in \Pi \begin{cases} h \equiv \bot\text{: } B \nsubseteq I \\ h \not\equiv \bot\text{: } B \subseteq I \Rightarrow l \in I \end{cases}$$

M is a minimal model of Π if, given the set of all models of Π: $M_1, \ldots, M_n, B_j, 1 \leq j \leq n$ such that M_j is a strict subset of M.

Models of programs represent interpretations of the program which include atoms that are supported by one or more rules of the program. A minimal model of a program is a model in which *only* supported atoms are included.

However, the above definition does not take into account negation as failure; in this case, rules of the program may contain negated literals which must not be in the interpretations of the program in order for the rule to be supported. In order to account for programs containing negation as failure, we define a reduct or transformation, often referred to as the *Gelfond-Lifschitz reduct* [73], as follows:

Definition 4.13. Let Π be a ground program. The Gelfond-Lifschitz reduct of Π with respect to an interpretation I where $I \subset HB_\Pi$, is the program Π^I containing rules $l \leftarrow B$ such that for all rules of the form:

$$l \leftarrow B, \mathbf{not}\ C \in \Pi, C \cap I = \emptyset$$

The reduced program includes all rules of the original program, omitting any rules which contain negated literals which are in the interpretation. The answer sets of a program are defined as follows:

Definition 4.14. A set of ground atoms I is an answer set of Π iff I is the minimal model of Π^I.

The uncertain interpretation of negation as failure gives rise to the possibility of more than one answer set, each of which is an acceptable solution to the program. It is this non-determinism in which the strength of ASP lies, where we are able to model problems that may have more than one solution [57]. We refer to the set of answer sets for a program Π as $\mathcal{A}_\Pi$.

Example 4.5. Consider the simple program Π:

$$a \leftarrow .$$
$$b \leftarrow a, \mathbf{not}\ c.$$
$$c \leftarrow a, \mathbf{not}\ b.$$

The Herbrand Base, HB_Π, of this program (the set of all atoms used in rules in the program) is the set of atoms $\{a, b, c\}$.

The program has the interpretations (similar to the power set):

$$\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\} \tag{4.2}$$

Given the interpretation $J = \{a, b\}$ and the Gelfond-Lifschitz reduct Π^J:

$$a \leftarrow .$$
$$b \leftarrow a.$$

The interpretation J is a model of Π, as the atoms in J are supported by both rules. This interpretation is also a minimal model with respect to Π^J as it includes only atoms supported by the program.

In contrast, given the interpretation $K = \{c\}$ and the transformation Π^K:

$$a \leftarrow .$$
$$c \leftarrow a.$$

The interpretation K is not a model of Π^K as it does not include the atom a which is supported by Π^K.

Finally, given the interpretation $L = \{a, b, c\}$ and the Gelfond-Lifschitz reduct Π^L:

$$a \leftarrow .$$

The interpretation L is a model of Π^L as it includes a but is not a minimal model as it includes the atoms $\{b, c\}$ which are not supported by Π^L. The same is also true for the empty set $\{\}$.

Therefore, the answer sets of the above program are the interpretations $\{a, b\}$ and $\{a, c\}$.

4.7 Domain Complexity

ASP is very expressive in a precise mathematical sense; for *AnsProlog* in its general form, allowing negation in rule bodies (but no disjunction in the rule heads), deciding whether a finite ground logic program has an answer set is NP-complete [130]. Furthermore, allowing disjunction in the rule heads, ASP can represent every problem in the complexity class Σ_2^{P} (and hence, deciding whether a disjunctive logic program has an answer set is Σ_2^{P}-complete). Thus, ASP is strictly more powerful than propositional satisfiability (known as SAT, which is NP-complete, one of the first problems in computational complexity theory for this to be proven [65]), as it allows for solving problems which cannot be translated to SAT in polynomial time (unless $\mathrm{P} = \mathrm{NP}$) [38].

However, while many of these problems are NP-complete in the worst-case, numerous instances are tractable in the general case and it is the solving times of real problems that is of interest. Due to the increasing efficiency of its solver tools, ASP is particularly suited to modelling difficult (primarily NP-hard) search problems.

For a detailed discussion of the complexity classes relating to *AnsProlog* and its associated subclasses, see Baral [16].

4.8 ASP Tools

A number of algorithms have been proposed for computing the answer sets of a logic program, leading to the development of numerous solver tools. Nearly all of the available solvers deal with ground programs, in which all variables have been removed, so the following section presents a short overview of the available grounding tools before we describe the main classes of solvers. For an informal specification of the core *AnsProlog* language accepted by the majority of the domain tools, see Appendix A (page 110).

4.8.1 Grounding Tools

As discussed in Section 4.5 (page 28), before computation an ASP program is *grounded* to remove variables, an instantiation process that creates copies of the rules for each usable value of each variable. The grounding process can be computationally-expensive [27, 115], in some instances even in proportion to the time cost of actually solving the program. This can be indicative of poorly-defined programs, where unnecessary modelling constructs or complex statements have a significant effect on the time taken to solve the program. Nevertheless, many modern grounding tools will attempt to perform a number of simplifications of the program during the grounding process, such as removing redundant rules [71, 171].

Two of the most common grounding systems are LPARSE [171] (developed alongside the SMODELS solving system) and GRINGO [67, 68, 71] (part of the CLASP family of ASP tools), while the DLV system [116] includes a grounding component as part of the front-end of the main system.

4.8.2 Solving Tools

The development of efficient ASP solver tools has increased rapidly over the past two years. Advances in the development of SAT-based and hybrid solver tools has pushed the boundaries for applications and benchmark competitions [49, 70]. A detailed comparison of solver tools is presented in a number of survey papers [76, 129].

The SMODELS System

The SMODELS system [141] was one of the first ASP solving system which included an efficient solving algorithm [140] and associated tools to assist the practical application of Answer Set Programming.

The SMODELS system consists of two programs: the SMODELS solver, which generates answer sets of ground programs; and the grounder LPARSE [164, 171], which accepts unground *AnsProlog* programs and generates a program in an efficient internal format for SMODELS.

The language accepted by LPARSE and the SMODELS system is referred to *AnsProlog*^{sm}, as it is an extension to *AnsProlog** and includes additional features such as cardinality rules, weighted atoms and weight constraint rules. For a detailed description of these features, see Niemela and Simons [142].

SMODELS is capable of reasoning efficiently with programs which contain large numbers of rules and atoms [165] and is based on the DPLL algorithm [44]. This is a backtracking search algorithm which exhibits tree-like resolution for deciding the satisfiability of propositional logic formulae in conjunctive normal form (known as CNF-SAT) [44].

Derivations of the SMODELS system include SMODELS-IE [25], a cache-optimised version; and SMODELS-CC [178], which applied an early form of clause learning.

SAT-based and Conflict-Driven Clause Learning Systems

ASP solvers, such as SMODELS and DLV, efficiently generate answer sets using carefully adapted heuristic algorithms designed for that purpose. In recent years, the study of complex reasoning and problem-solving systems has lead to the definition of a number of efficient algorithms for solving the more general Boolean (propositional) satisfiability problem (known as SAT) [51, 136]. These advances led to the development of ASSAT [122], CMODELS [120], SMODELS-CC [178], CLASP [69] and SUP [119] for generating answer sets using SAT solving techniques or using hybrid approaches.

The early SAT-based systems (ASSAT and CMODELS) translate an *AnsProlog* program (with some constraints on the program structure) into a propositional satisfiability problem containing a set of Boolean formulae which describe the constraints on the atoms in the program. These satisfiability problems are solved directly using a SAT solver which yields the solutions and which may in turn be translated back into answer sets of the program. However, more recent systems such as CLASP are based on techniques from constraint solving and could be regarded as neither a pure ASP nor a pure SAT-based solver, as there is no explicit conversion between representations [69].

Modern SAT-based ASP solvers, utilising a number of techniques derived from state of the art SAT solving systems (such as conflict-driven clause learning [69]), have been shown to perform competitively when compared to the conventional backtracking solvers; often outperforming their conventional counterparts [119, 120] by an order of magnitude in certain cases.

The DLV System

The DLV system [52] originated as a system for reasoning in *Datalog*, an extension of Answer Set Programming which allows exclusive disjunction (logical OR) in the head of the rules. As well as supporting disjunction, DLV also handles a large subset of the

AnsProlog language and is capable of solving problems involving large numbers of rules efficiently [116]. However, it accepts as input a different syntax from the widely-accepted LPARSE/SMODELS format. Numerous extensions to the DLV system exist, including an SQL front-end [116], but they are out of scope for this thesis.

Other Systems

A range of other solvers for ASP programs exist, including DERES [30], SURYA [134] and NOMORE++ [7], which are either adaptations of existing solver frameworks or hybrid solving systems. There has also been recent development in incremental answer set solving, with ICLINGO [67], a stateful implementation of the GRINGO grounder and CLASP solver; along with early versions of solvers than ground on the fly, such as ASPERIX [114]. Recent work in the optimisation of ASP programs and in particular, removing redundant rules from programs [94], has also improved the performance of ASP solvers.

There has also been work in the field of distributed ASP solving, particular with PLATYPUS [81] (based on the SMODELS system) and CLASPPAR [54], an early distributed version of CLASP. Distributed solving [150] and the use of Beowulf systems in ASP solving has demonstrated the validity of the approach [24, 149], along with preliminary work in parallel grounding [27].

4.9 Applications of ASP

ASP has been successfully applied to various application areas outside of the traditional domains of planning and diagnosis [13, 52, 121] (with the most notable implementation of decision support systems for the NASA Space Shuttle [143]); including software engineering [145], instruction scheduling [108], program analysis [181], automatic music composition [21, 22], e-tourism [89], evolutionary history of languages [58], biological networks [72], phylogenetics [55], haplotype inference [56], multi-agent systems [28, 45, 46], security engineering [75] and cryptography [47, 86].

The *Working Group on Answer Set Semantics* (WASP), a European Commission Fifth Framework Programme (FP5)-funded project (from 2002-2005), further developed research in the Answer Set Programming formalism and related tools [59]. The wide range of domains to which ASP has been applied demonstrates its versatility and applicability in modelling complex real-world problems.

4.10 Summary

In this chapter we presented an overview of the Answer Set Programming (ASP) paradigm and its underlying formalism *AnsProlog*, its origins and development within the wider field of logic programming and non-monotonic reasoning and its relationship to Prolog. Furthermore, we described its clear syntax and semantics and introduced the state of the art domain tools for generating solutions (answer sets) of programs. We also presented a wide range of successful applications of ASP technology in modelling real-world problems.

In the next chapter we present our application of Answer Set Programming: the TOAST superoptimising system, which generates provably optimal code sequences for modern machine architectures.

Chapter 5

TOAST: Total Optimisation using Answer Set Technology

The worthwhile problems are the ones you can really solve or help solve, the ones you can really contribute something to.

Richard Feynman

5.1 Introduction

As was identified in Chapter 2, none of the existing approaches to optimising code specifically aims for optimality from the outset. Superoptimisation, introduced in Chapter 3, addresses this problem by providing an approach that can generate optimal code sequences for a particular goal function on a specific machine architecture. Existing superoptimising implementations, such as the GNU Superoptimizer (GSO) [79] and the Stanford superoptimiser [14], are able to generate optimal sequences under certain constraints, but there are a number of caveats with their approaches, especially regarding verification of the equivalence of code sequences.

In this chapter, we introduce TOAST, the *Total Optimisation using Answer Set Technology* system, a provably optimal code generation system that applies superoptimising techniques to generate optimal sequences for acyclic, integer-based code. The TOAST system utilises Answer Set Programming, introduced in Chapter 4, as an expressive modelling language and efficient computational framework.

We first describe the motivation for the design of the TOAST system, then describe its main components, illustrated by optimising sequences for the MIPS R2000, a 32-bit RISC architecture. We then discuss key design issues and present a model for validating the system, along with examples of searching for candidates sequences and verifying that these sequences are indeed optimal. We demonstrate that the TOAST approach to generating provably optimal code sequences is achievable and scalable for real code sequences on 32-bit architectures of up to five instructions long. This is actually a significant result, due to the average size of basic blocks in code being on average between 5-6 instructions long [84, 103]. By superoptimising real code sequences of these lengths, it is possible to extend the system and apply the technique to programs of arbitrary length.

An initial proof of concept design for the TOAST system was first presented in Brain et al. [24], with some benchmark results for optimising code sequences presented in Crick et al. [37].

5.2 Motivation

As discussed in Chapters 1 and 2, new strategies for creating efficient and cost-effective compiler tools and hence new strategies for optimising code for modern architectures is required [82]. Being able to generate efficient code for an architecture is of huge importance; this is the prime motivation for the development of the TOAST system.

Aside from the inclusion of results from GSO into a specific architecture port of GCC [79], the lack of uptake of the existing implementations has prevented the further development of superoptimising techniques. Their approaches have not scaled well for real code sequences, especially with proving the equivalence of two code sequences. Another of the key issues is to do with guaranteeing the optimality of the sequences generated. All of the existing implementations perform a representative test or time-out during equivalence verification. While this is a cautious approach to verification – and in doing so, makes the problem more tractable, as they invariably discard the more troublesome sequences to verify – it is possible to construct cases which could pass a representative test but fail a full equivalence test (this process will be discussed further in Section 5.6). By timing out in this way, there is a risk of discarding potentially optimal sequences. In contrast, the TOAST system performs a full verification stage on all generated sequences.

The expressibility of ASP and its clear applicability to modelling real-world problems,

along with the availability of efficient domain tools, are the key reasons for its application within the TOAST system. Since its inception, it has been regarded as the computational embodiment of non-monotonic reasoning and a primary candidate for an effective knowledge representation tool. It is a burgeoning research area [59] and has been successfully applied to a wide range of domains (as shown in Section 4.9, page 40). This significant application of ASP technology in disparate domains has had a positive effect for tool development, especially over the past two years. In the TOAST system, we utilise off-the-shelf, open source solver tools, with our main criterion for use being the correct output in the fastest time.

However, ASP is not the only knowledge representation or declarative logic language with real-world applications. Other paradigms such as constraint logic programming, linear (integer) programming or even propositional satisfiability-based representations could have been utilised for the TOAST system, but the expressiveness, clear modelling semantics and wide availability of efficient domain tools for ASP push it to the forefront as an efficient language for modelling real-world problems.

The use of logic and declarative techniques for compiler-related problems has a long history: for example, using logic programming for compiler development [177] (especially Prolog [32, 118, 179]), along with declarative techniques for analysis and optimisation [61], proving the correctness of optimisations [109, 139], register allocation [77] and instruction scheduling [128, 182]. In particular, ASP has been applied to program analysis [181] and more recently, multi-core instruction scheduling [108], with some success.

5.3 Architecture Overview: MIPS R2000

In this section we give an overview of the MIPS R2000 architecture, one of the test architectures for the TOAST system, which is used for the benchmarking tests in this chapter.

The MIPS (originally an acronym for *Microprocessor without Interlocked Pipeline Stages*) is a load/store reduced instruction set computer (RISC) architecture. The MIPS architecture family has had broad application in embedded systems; in the late 1990s a third of all RISC processors were MIPS-based [148]. The early MIPS architectures were 32-bit, while later versions were 64-bit; the current revisions are the MIPS32 and MIPS64 [99].

The first commercial MIPS CPU model, the R2000, was announced in 1985. It added

multiple-cycle multiply and divide instructions in a somewhat independent on-chip unit. The R2000 can be used in either big-endian or little-endian mode. It has thirty-two 32-bit general purpose registers, but no condition code register (as the designers considered it a potential bottleneck), a feature it shares with the AMD 29000 and the DEC Alpha architectures. Also, unlike other registers, the program counter is not directly accessible [99]. The MIPS design uses triadic addressing, with six bits of the 32-bit word for the basic opcode; the rest may contain a single 26-bit jump address or it may have up to four 5-bit fields specifying up to three registers, plus a shift value combined with another six bits of opcode; another format, specifies two registers combined with a 16-bit immediate value. This allows the CPU to load up the instruction and the data it needed in a single cycle [99].

The R2000 also had support for up to four co-processors, one of which is built into the main CPU and handled exceptions, traps and memory management, while the other three are left for other uses, such as floating point operations.

The MIPS R2000 was chosen as a test architecture for the TOAST system as it was possible to concisely model a significant proportion of its instruction set (even though multiple and divide are modelled as multiple-cycle instructions) and that it is an example of a generalised 32-bit RISC architecture. Also, the availability of a mature MIPS simulator (SPIM [148]) that can read and execute assembly language programs, made it an effective initial validation architecture for the TOAST system.

The architectural description for the MIPS R2000 used in the TOAST system can be found in Appendix C (page 117).

5.4 System Overview

5.4.1 Introduction

The TOAST system consists of modular interacting components that generate *Ans-Prolog* programs and parse answer sets, with a controlling interface that applies these components to generate a superoptimised version of the original code sequence. Information is passed between components either as fragments of ASP programs or in an architecture-independent, assembly language-like format.

Input to the TOAST system is a sequence of instructions in an internal format, with the output either a shorter, optimal program, or no shorter optimal sequence exists. The format of TOAST input programs consists of declarations of inputs, instructions

and outputs, as represented by the Extended BNF description given in Listing 5.1; an example TOAST input program is given in Listing 5.2.

```
toastprogram = { input } , { instruction }, { output } ;
input = "in: " , { variable , immediate , mixed , flag } ;
instruction = "inst: " , instructionname , { instructionarg } ;
outputs = "out: " , { variable , immediate , mixed , flag } ;
variable = "v" , number ;
immediate = "i" , number ;
mixed = { "v" , binarydigit } ;
flag = "f+" , flagname | "f-" , flagname | "?" , flagname ;
flagname = ? any valid flag name ? ;
instructionname = ? any valid machine instruction ? ;
instructionarg = "i" , number | number ;
number = digit , { digit } ;
digit = binarydigit | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
binarydigit = "0" | "1" ;
```

Listing 5.1: Description of TOAST program input format in Extended BNF

In addition to the Extended BNF description, the notation presented in Table 5.1 gives an overview of the input and output encoding, specifically how to represent values and also how to set and check the architectural flags. For example, `01vvv001` would represents an eight bit pattern with three variable bits that would be randomly assigned, while an architectural flag example would be `f+carry`, which sets the carry flag to 1 on input.

Notation	Description
`v`	Variable, followed by number of bits
`i`	Immediate, followed by value in two's-complement format
`m`	Mixed, bit pattern in which specified bits are fixed or variable
`f+`	Flag (set to 1 on input), then flag name
`f-`	Flag (set to 0 on input), then flag name
`f?`	Flag (set on output), then flag name

Table 5.1: TOAST program input format notation

The format of the instruction arguments is as follows: `i0` refers to the hardwired zero (if it exists for that architecture), `ix` refers to the x^{th} input (indexed from one), while an integer `y` refers to the output of the y^{th} instruction in the sequence.

ASP solvers are used as 'black-box' tools to generate solutions to the programs produced by the TOAST system. We first generate an *AnsProlog* encoding of the input program, its instructions and the number and type of inputs and outputs. This program is then used as a starting point for searching for candidate sequences of shorter length, start from length one. The answer sets (i.e. models) produced by the solvers are the candidate sequences encoded in *AnsProlog*. This set of candidates is pruned

using a number of heuristics and each candidate is tested for equivalence to the original sequence; first using a fast probabilistic test and if passed, a full equivalence test over all inputs. If the sequence is equivalent, no answer sets will be returned from the solver; essentially the *AnsProlog* program models the verification as *show me the inputs on which these two programs differ*. Hence, no answer sets returned means they do not differ on any input and are equivalent. The structure and key components of the TOAST system, presented as a process for an example program, is shown in Figure 5-4 (page 63).

```
in: v32
in: v32
inst: add i1 i2
inst: sub 1 i1
out: v32
```

Listing 5.2: Example program in TOAST input format

While previous superoptimising implementations have been written in a machine-specific assembly language [131] or C [14, 79, 95], the TOAST system has been developed using Perl. This is due to the observation that most of the actions performed within the system are fundamentally text processing: manipulating programs and instructions, parsing answer sets and outputting sequences. While there may be performance ramifications in comparison to using assembly language or C, the text processing capabilities of Perl are of huge benefit. Compared to previous superoptimising implementations, we have no requirement or dependencies on efficiently encoded goal functions (as in GSO), or require the creation of large data structures to map the search space (as in the Stanford system). In the TOAST system, all of the computation of solutions is delegated to the ASP solver tools.

5.4.2 Architectural Modelling

The architecture descriptions used in the TOAST system define the mappings from the architecture-specific assembly language syntax to the TOAST internal instruction format. *AnsProlog* is used to model the functionality of the integer processing unit of the target processors; this is usually a one-to-one mapping. The expressive nature of ASP enables simple and concise modelling of the properties of each architecture, such as the bit-level semantics of the instructions, while also allowing the modelling of complex constraints. The majority of the modelling is at the bit level; for example, *AnsProlog* rules that relate input bits of an instruction to the output bits.

The TOAST system adheres to the following architecture modelling rules, but they can be encapsulated by *only instructions that change registers and/or condition codes are modelled*. Furthermore:

- Floating point instructions are not modelled; this is due to the significant complexities of representing floating point numbers and their arithmetic operations in the numerous floating point systems, along with handling special values (for example, signed zero, subnormal numbers, infinities and NaNs [2]), conversions and rounding modes. This is further compounded by the problems of proving the equivalence of two floating point values.

- Any instruction that touches memory, either altering or putting addresses into registers, is not modelled. This is because the TOAST system currently has no model of the memory system (apart from the machine registers) because of the complexities with modelling large amounts of memory.

- Higher privilege instructions are not modelled, as these are an unlikely source of optimal sequences for normal programs: if they are being requested, a hardware interrupt has occurred and they are not directly executed by the user. Similarly, it does not model anything that could throw an exception.

- Co-processor operations and vector operations are not modelled, as they also depend on precise modelling of the memory layout. However, if the co-processor has its own set of registers then it can be modelled as a separate processor and integrated. Vector operations can be modelled as single operations and then converted.

- A general modelling decision: anything that does not create a value (for example, `nop`, `jmp`, etc) should not alter value for that time step. While this is common sense, it is explicitly modelled.

The architecture description file also defines which flags are available for that specific processor (see Appendix C, page 117), via a space-separated list or declared as 'none'. All flags are undefined at the start of execution and are governed by simple inertia rules: they maintain their value unless changed. To use a flag, the `flagStatus(C,T,F)` *AnsProlog* literal must be referenced. To change the value of a flag, set its new value (i.e. `flagStatus(C,T,F)` or `-flagStatus(C,T,F)`) and then set `flagChanged(C,T,F)`. Both literals must be set or the inertia principle will still

hold and create a contradiction if the flag's status has changed. For an overview of the *AnsProlog* literals used within the TOAST system, see Appendix B (page 112).

The description is used to generate a list of which instructions are available for a given architecture, along with general information, such as the word size and the availability of a zero register (a special hardware register that ignores anything written to it and will always return zero when read). The TOAST system currently supports the following architectures: MIPS R2000, SPARC V7 and SPARC V8 (see Appendix C, page 117), along with a number of test architectures adapted from the MIPS R2000 instruction set.

The instruction sequence itself is represented as a series of facts, or in the case of searching, a set of choice rules in *AnsProlog*. 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 classical 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 5.4 (page 50). 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. None of the programs generated within the TOAST system requires disjunction, aggregates or any other non-syntactic extensions to the answer set semantics (as discussed in Chapter 4, page 24).

```
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 5.3: *AnsProlog* encoding of TOAST flow control rules

The instruction library describes properties of the instructions, such as whether they are unary or binary, symmetric or asymmetric in their arguments, along with an *AnsProlog* description of the semantics of the instruction. A more complex example is given in Listing 5.5 (page 51) of an arithmetic ADD instruction, which demonstrates the relationship between the first bit and the following bits and how the carry is handled.

Flow control rules define which instruction will be 'executed' at a given time step by controlling the program counter (`pc`) literal. As *AnsProlog* programs in the TOAST system may need to simultaneously model multiple independent code streams (for example, when trying to verify the equivalence of two sequences of code), all literals are tagged with an abstract property named *colour*. The inclusion of the `colour(C)`

```
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 5.4: *AnsProlog* encoding of the logical AND (`land`) instruction

literal in each rule 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 in generating the sets of rules is negligible. An example encoding of a flow control rule is given in Listing 5.3, while a description of the important literals relating to the `colour` property can be found in Appendix B (page 112).

By using basic instruction descriptions and *AnsProlog* to model the semantics of these instructions, it is possible to rapidly model new architectures and their instructions sets, along with making it simple to tweak and amend the semantics of instructions. Clearly, an understanding of the target architecture is required, but this means it is possible to model a new architecture for use within the TOAST system by using the information available in an architecture reference manual. Porting to a new architecture is dependent on how many of the instructions in the new architecture have already been modelled within the TOAST system (and whether you are required to model any significant non-standard features or semantics).

5.4.3 Components

In the following section, we introduce the key components of the TOAST system (see Figure 5-4 for a process diagram of the system, page 63), their functionality and how they interact. Each of the components are separate programs that are utilised within the TOAST system to generate optimal sequences.

As described in Section 5.4, the input to the TOAST system is a program in the TOAST-specific format which encodes the instruction sequence and information about its inputs and outputs. The output of the TOAST system is either a shorter optimal sequence, or nothing if the sequence cannot be optimised further.

```
% First bit
-value(C,T,0) :- istream(C,P,add,R1,R2,none), pc(C,P,T), -value(C,R1,0),
    -value(C,R2,0), colour(C), position(C,P), time(C,T), register(R1), register(R2).
-value(C,T,0) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,0),
    value(C,R2,0), colour(C), position(C,P), time(C,T), register(R1), register(R2).
value(C,T,0) :- istream(C,P,add,R1,R2,none), pc(C,P,T), not -value(C,T,0),
    colour(C), position(C,P), time(C,T), register(R1), register(R2).
additionCarry(C,T,1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,0),
    value(C,R2,0), colour(C), position(C,P), time(C,T), register(R1), register(R2).
-additionCarry(C,T,1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), not
    additionCarry(C,T,1), colour(C), position(C,P), time(C,T), register(R1),
    register(R2).

% Subsequent bits
value(C,T,B) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,B),
    value(C,R2,B), additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
value(C,T,B) :- istream(C,P,add,R1,R2,none), pc(C,P,T), -value(C,R1,B),
    -value(C,R2,B), additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
value(C,T,B) :- istream(C,P,add,R1,R2,none), pc(C,P,T), -value(C,R1,B),
    value(C,R2,B), -additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
value(C,T,B) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,B),
    -value(C,R2,B), -additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
-value(C,T,B) :- istream(C,P,add,R1,R2,none), pc(C,P,T), not value(C,T,B),
    colour(C), position(C,P), time(C,T), bit(B), B != 0, register(R1), register(R2).

additionCarry(C,T,B+1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,B),
    value(C,R2,B), additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
additionCarry(C,T,B+1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), -value(C,R1,B),
    value(C,R2,B), additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
additionCarry(C,T,B+1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,B),
    -value(C,R2,B), additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
additionCarry(C,T,B+1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), value(C,R1,B),
    value(C,R2,B), -additionCarry(C,T,B), colour(C), position(C,P), time(C,T),
    bit(B), B != 0, register(R1), register(R2).
-additionCarry(C,T,B+1) :- istream(C,P,add,R1,R2,none), pc(C,P,T), not
    additionCarry(C,T,B+1), colour(C), position(C,P), time(C,T), bit(B), B != 0,
    register(R1), register(R2).
symmetricInstruction(add).
```

Listing 5.5: *AnsProlog* encoding of the arithmetic `add` instruction

`toast` - the control program

input: program
output: program (*optimal*) or $\emptyset$

The `toast` program is the controlling interface to the TOAST system, enabling option-setting (such as fine-tuning of the search heuristics and verification options). The main control flow searches for candidate sequences, prunes the set of candidates and then verifies which of the candidates are equivalent to the original sequence. An important emphasis is on pruning as many sequences during the search phase as possible, in order to reduce the number of candidates to verify.

The key observation underlying the design of the TOAST system is that any super-optimised sequence will necessarily be returned by using `search` on the appropriate instruction length. However, not everything that `search` returns is necessarily a correct answer; due to the initial search constraints, it is possible that there are a number of invalid candidate sequences found. Thus, to remove these invalid candidates, the front end generates further search constraints from the input instruction sequence. Instruction sequences of length one, up to one less than the length of the input sequence, are then searched sequentially. If candidates are found, another constraint set is generated and the same length searched again. The two results sets are intersected, as any correct sequence must appear in both searches. 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 stabilises. `verify` is then used to check the candidates for equivalence to the original input sequence. A representative pre-verification test is performed before a full verification test, using a selected set of test vectors. The output of `toast` is an optimised version of the input program, if one exists.

`findPath`

input: program
output: paths

`findPath` generates a list of possible execution paths (if one exists) through the input program. As the TOAST system currently only supports straight-line code, without explicit branches or loops, this will return a comma-delimited list of integers representing the possible execution order of instructions in the input program. Example output from `findPath` for the program given in Listing 5.2 would be `1,2`, which represents the first instruction in the program, followed by the second instruction.

pickPath

input: path, program
output: *AnsProlog* vectors

`pickPath` generates a set of inputs (referred to as *input vectors*) in *AnsProlog* for the input program that will follow the instruction path generated from `findPath`. These input vectors are a set of selected bit values for all of the inputs in the original program. For example, if a program has three 32-bit inputs, then `pickPath` will generate three 32-bit binary values, picking a 0 or 1 for each bit. This gives an initial set of values for searching for candidate sequences. At this stage it is also possible to explicitly select difficult edge cases and pick vectors for these; for example, picking a set of values on a boundary, such as all zeros or all ones, to prune the search space of clearly redundant sequences.

execute

input: *AnsProlog* vectors, program
output: *AnsProlog* constraints

The `execute` program emulates running the input instruction sequence using the generated input vectors from `pickPath`, producing constraints in *AnsProlog* that describe the original instruction sequence's outputs. These constraints are used to bound the initial search space.

search

input: search space, *AnsProlog* vectors, *AnsProlog* constraints
output: program fragments

By using the input vectors and constraints (essentially start and end values for the original input program), it searches for all instruction sequences of a given length (the *search space*) that produce the correct output for the given input values. A number of heuristics have been developed to prune the initial search space, as certain sequences can generate large numbers of candidate sequences on initial searches. Code sequences that are used in an incorrect fashion (for example, sequences that discard previously calculated values, or instructions with invalid arguments) are discarded, hence avoiding a form of runtime error; we can also assert that every instruction must contribute to the output, otherwise it may be possible to have redundant instructions in the sequence. We can also do the same with asserting that every input must be used, along with every instruction's output. While the application of these heuristics does not affect the

outcome of the search, it can provide significant savings in runtime.

`searchCut`
input: instructions
output: *AnsProlog* constraints

`searchCut` generates extra constraints to append to those accepted by `search` to further trim the size of the search space. After the first search, it is possible to search within the original results rather than performing another full search. This is an obvious step, because any optimal sequence would have to be contained within this original search (because by definition, it would have to match the tuple of every possible input/output value).

`verify`
input: program, program, vectors*
output: Boolean

If candidate sequences are found during the search, then they have to be verified for full equivalence to the original sequence over all possible input values. In certain cases, it is possible for a large amount of candidate sequences to be generated that are not pruned during the search phase, so the following verify processes are performed:

- *pre-verify*, a fast heuristic that uses a directed set of input vectors to perform a representative verify on the two sequences. As mentioned previously, it is possible to explicitly select difficult edge cases and pick specific vectors for these; for example, picking a set of values on a boundary, such as all zeros or all ones. If pre-verify returns false, then the candidate is discarded (since it is definitely not an optimal sequence); if true, then a full verify must be performed to prove full equivalence.
- A *full verify* tests if two sequences are equivalent for all input values. If they are not equivalent, it is possible to output a set of vectors representing values on which they differ, in a suitable form for `execute` and `search`.

5.5 Experimental Results

Superoptimisation naturally decomposes into two discrete tasks: searching for candidate sequences and then verification of the equivalence of these candidates to the

original sequence. In this section, we present benchmarks for these two main tasks of the TOAST system for the MIPS R2000, a 32-bit RISC architecture. All tests were run on quad-core Intel 2.8GHz Xeon E5462 processors with 32GB RAM, running a variant of Scientific Linux. Programs were grounded using GRINGO and tested with the following five solvers: CLASP, CMODELS, SMODELS, SMODELS-IE and SUP; all tools were built in 32-bit mode (this is significant for a number of reasons, relating to the size of addressable memory, but mainly because a number of ASP tools do not currently build as 64-bit programs).

5.5.1 Searching

The `sequence5` search test, as given in Listing 5.6, attempts to find shorter optimal sequences for a five instruction program, with two 32-bit inputs. This sequence was selected as an example of a sequence that cannot be superoptimised (i.e. it is already optimal), giving an approximate ceiling on the performance of the system. Benchmark times for the `sequence5` test for the five chosen solvers are given in Table 5.2; solver timeouts occurred after 240 hours.

```
! input 1 in %i1
! input 2 in %i2
and %l1 %i2 %i2
add %l2 %i1 %l1
add %l3 %i1 %l2
add %l4 %i1 %l3
sub %o1 %i0 %l4
! output in %o1
```

Listing 5.6: `sequence5` search test for MIPS R2000

The timings plotted in Figure 5-1, confirm that search times increase at a near exponential rate as the sequence length increases. The TOAST system is able to search over sequences of five instructions in approximately six hours, but this is dependent on the solver used (discussed further in Section 5.5.3).

5.5.2 Verifying

The `argredundancetest` verify test checks to see if a non-trivial redundant argument is optimised away, reducing the three instruction sequence to one instruction. In these tests, we amended the bit-level modelling of the input programs to demonstrate

	Search sequence length				
Solver	**1**	**2**	**3**	**4**	**5**
clasp-1.2.1	0.23	1.83	179.00	5191.48	20268.21
cmodels-3.79	0.36	6.12	1006.00	4244.83	21176.82
smodels-2.33	0.24	6.58	5578.63	t/o	t/o
smodels-ie-1.0.0	0.18	4.91	2115.37	t/o	t/o
sup-0.4	0.46	2.18	177.24	5516.97	22942.46
Atoms	853	1411	2098	2941	4003
Rules	47925	130956	259223	442589	712166

Table 5.2: Timings (in sec) for `sequence5` search test on MIPS R2000

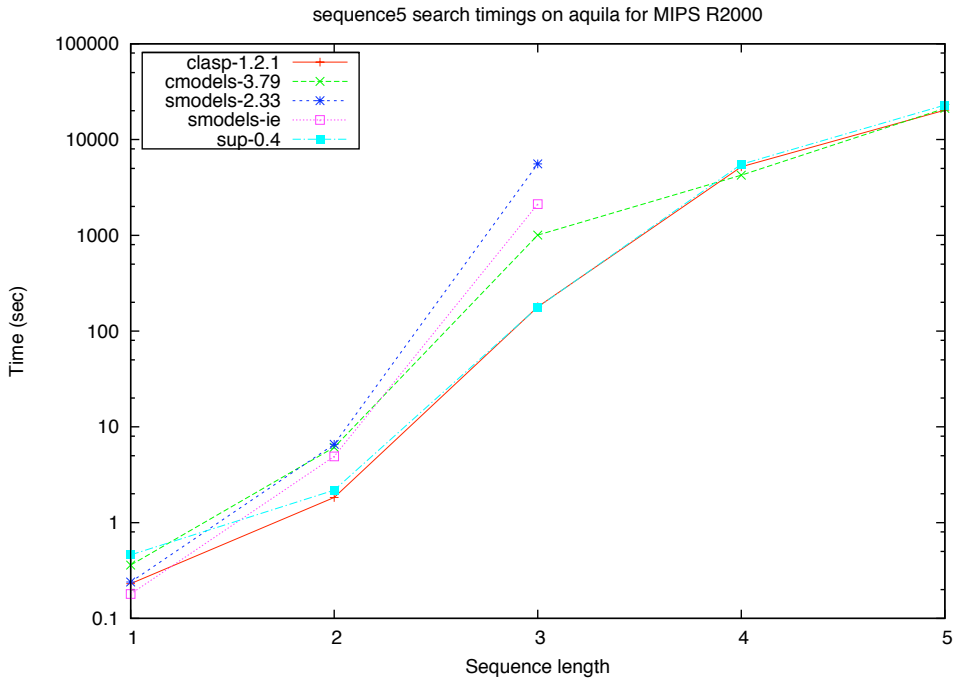

Figure 5-1: Plot of `sequence5` search test times (in sec) on MIPS R2000

the scalability of the TOAST system in verifying sequences on 8-bit, 16-bit, 32-bit and 64-bit architectures. Benchmarks for the five solvers is given in Table 5.3.

The timings plotted in Figure 5-2, confirm that verify times also appear to increase at an exponential rate as word size increases, as expected. The TOAST system is able to verify most 32-bit programs in less than one second, but again this is dependent on the type of solver used (again, discussed further in Section 5.5.3).

```
in: v32
in: v32
in: v32
inst : lxor i2 i3
inst : lxor 1 i3
inst : add i1 2
out: v32
```

Listing 5.7: `argredundancetest` verify test for MIPS R2000

	Program word size			
Solver	**8-bit**	**16-bit**	**32-bit**	**64-bit**
clasp-1.2.1	0.12	0.05	0.15	2.00
cmodels-3.79	0.11	0.06	0.39	0.82
smodels-2.33	0.14	11.21	-	-
smodels-ie-1.0.0	0.06	11.38	-	-
sup-0.4	0.24	3.16	-	-
Atoms	922	2314	7402	26858
Rules	1643	4915	17219	64803

Table 5.3: Timings (in sec) for `argredundancetest` verify test on MIPS R2000

5.5.3 ASP Tool Benchmarking

In this section we analyse the search and verify results with respect to ASP tool performance.

Grounding Tools

The grounding process is an overlooked component of solving, but in certain cases can represent a large proportion of the overall solving time. Grounding becomes more of an issue in scenarios when you need to perform a large number of solves; for example, when a TOAST run generates a large number of candidate sequences that need to be verified. From the benchmark timings in Table 5.4 (and plotted in Figure 5-3) for the two most common grounding tools LPARSE and GRINGO in comparison to the associated solver results in Figure 5.3, it takes at least an equivalent amount of time to ground as it does to solve a 32-bit program. GRINGO does appear to scale at near linear time for increased bit size compared to LPARSE, which becomes increasingly more expensive. Modern grounders perform simple optimisations and remove redundant parts of a program, but in certain cases a naive fast grounding option would be more suitable. Grounding has been neglected as a research and development area in favour of

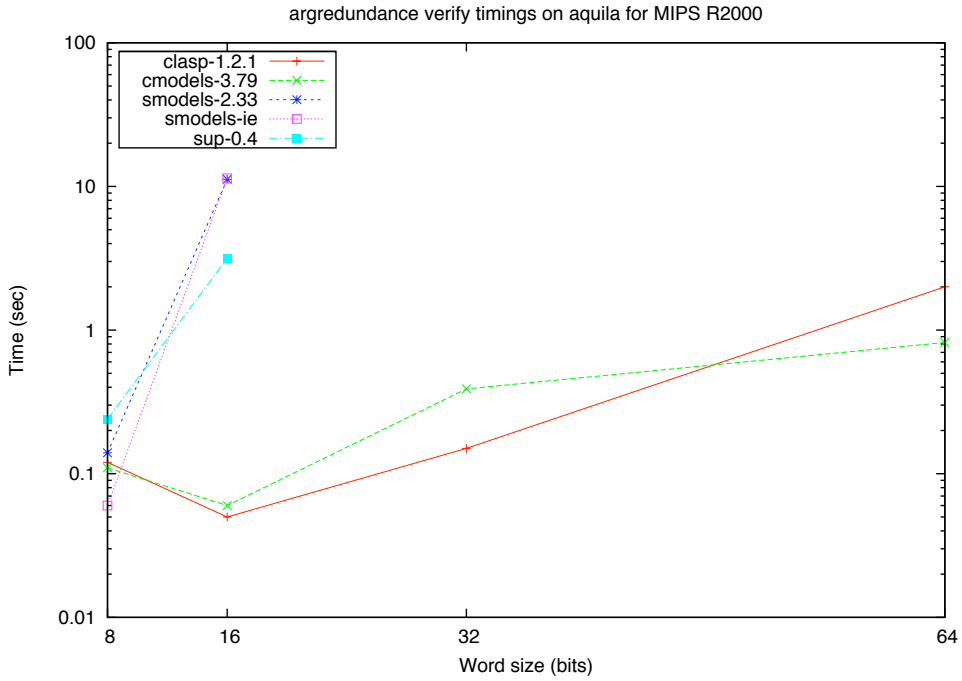

Figure 5-2: Plot of `argredundancetest` verify test times (in sec) for increasing bit size on MIPS R2000

developing more sophisticated solving algorithms, but there are a number of domains that require more efficient grounding solutions, including TOAST. This is increasingly apparent in certain applications of the TOAST system, as presented in Chapter 7.

	Program bit size			
Grounder	**8**	**16**	**32**	**64**
gringo-2.0.3	0.06	0.12	0.13	0.13
lparse-1.1.1	0.20	0.38	0.74	1.37
Atoms	1524	2820	5412	10596
Rules	3580	7164	14332	28668

Table 5.4: Timings (in sec) for `verifytest1` grounding tests on MIPS R2000

Solver Tools

The solver results presented in Table 5.3 indicate that there is a large gap between the two main classes of solvers used: the well-established solvers based on the SMODELS

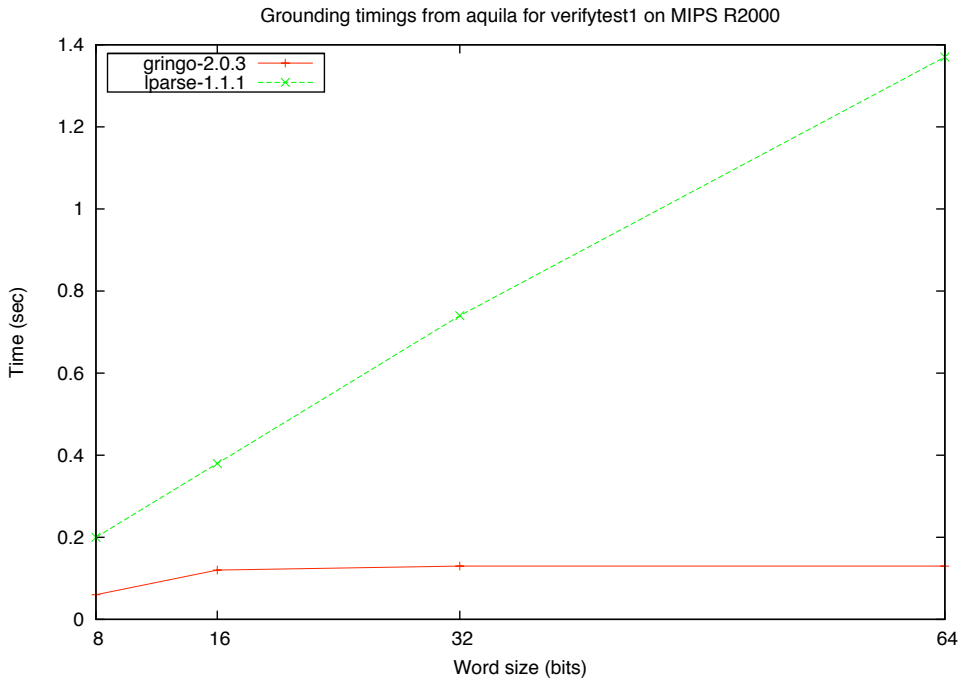

Figure 5-3: Plot of grounding times (in sec) for increasing bit size on MIPS R2000

algorithm and the more recent SAT-based and clause learning solvers, such as CLASP, SUP and CMODELS. The performance of the latter class of solvers on TOAST-specific problems are orders of magnitude faster than the SMODELS-based tools. As can be seen in Table 5.3 (and also in results presented in Chapters 6 and 7), we are unable to generate results for verifying 32-bit programs with the SMODELS-based tools, on any architecture. In fact, the upper bound for this class of solver is greater than 250 hours of compute time. This has influenced the solver choice for use within the TOAST system.

5.6 Discussion

As we have presented in this chapter, superoptimisation decomposes into the two main tasks of searching for candidate sequences and then verifying their equivalence. In certain cases, it is possible to generate a large number of candidates during the search phase (which is problematic during the computationally-heavy verification phase), but none of the implemented heuristics have the potential of discarding any valid se-

quences.

Alongside the 32-bit architectures that we have modelled (MIPS R2000 and SPARC V7/V8), pseudo 8-bit and 16-bit architectures (derived from the MIPS R2000) have been modelled to demonstrate the complexity curve with respect to increasing word size. This required manipulation of certain instructions to create 8-bit and 16-bit specific versions (notably logical shifts, but generally operations that are concerned about the actual length of the bit sequences). This has provided an insight to how the TOAST system scales with increasing architecture word size. TOAST is able to search for and verify sequences on 32-bit architectures, while searching is more dependent on the number of instructions in the input sequence. On an n-bit architecture, the raw search space is $2^{\text{inputs}*n}$, so TOAST is currently able to search over sequences of up to five instructions in an acceptable time. However, as discussed earlier, this is a reasonable limitation and actually a significant result, with empirical studies [84, 148] showing that the average size of basic blocks is between 5-6 instructions long. Hence, TOAST can be applied to optimising real-world code sequences.

The 8-bit and 16-bit test architectures were used to verify and validate the search process to ensure that actual candidates are not discarded or invalid sequences included. Using the 8-bit architecture, the TOAST system was validated to ensure it met the original design specification. By using an exhaustive search without any pruning or heuristics, we were able to see if all possible instruction sequences were generated. This was validated against the architecture description. Each search heuristic was then tested to ensure that sequences were not inappropriately added or removed; again, this was validated against the architectural model.

The use of the pre-verify heuristic is a key part of the verification model of the TOAST system. As with the existing superoptimising implementations, especially GSO [79], a representative verify test enables a fast check of whether sequences are equivalent. However, we do not overlook a full verification strategy, in contrast to other implementations. If a sequence is discarded by pre-verify, it is definitely not a valid sequence, but the reverse is not true: if a sequence passes the pre-verify test, there is no guarantee that this is equivalent. A full verify must be performed to guarantee full equivalence for all input/output values. Empirical tests [37] from repeated runs of the TOAST system has shown that while we have never encountered a sequence that has passed pre-verify but failed a full verify, it would be trivial to construct one. In fact, it would be feasible to construct a program to get the TOAST system to generate these programs. For example, since the heuristic only selects values for the middle bits of a sequence, a pair of instruction sequences that differ on one input and only use the

lower 25 bits (assuming a 32-bit word), prefixing both with shift left 2 and shift right (arithmetic or zero fill) 7 would give a pair that would pass pre-verify with probability of $1 - \frac{1}{2^{25}}$. It may also be possible to utilise the TOAST system to generate sets of these sequences. However, whether these are likely to appear in real code sequences remains to be seen. Nevertheless, by only performing representative testing (as in Massalin's original superoptimiser and GSO) or timing out after a fixed period (like the Stanford superoptimiser) means there is a chance that you either validate a non-optimal sequence or potentially discard an optimal sequence. With no guarantee of equivalence there is always the requirement to hand-check sequences. By design, this is not possible in the TOAST system.

As stated earlier, the architectural model is complete with respect to the instructions that we have chosen to represent; in this case no floating point operations, privileged instructions or anything that allows exceptions. While this is a simplification of the complete functionality of the architecture, it is a justifiable one due to the complexities of proving equivalences in floating point operations and the problems of manipulating privileged and exception-raising code sequences. The caveat of this choice is that there may exist potential improvements or amendments to the model that could allow more optimal sequences. This issue is discussed further in the future work section in Chapter 8.

With the goal-directed superoptimisers like GSO means that while they may be quicker in finding certain sequences, they are not able to accept any arbitrary code sequence like the TOAST system. This means that the burden is on the user to efficiently encode new goal functions, with a requirement to understand the underlying nature of the problem. This is not the case for TOAST; while there is a measure of translating and encoding required for the architectural descriptions, a potential user would only need to know the TOAST program input format and how to run the system.

5.7 Summary

In this chapter we have presented the TOAST: *Total Optimisation using Answer Set Technology* superoptimising system, along with its main components, and have shown the following:

- The TOAST system is a practical superoptimising toolchain that guarantees full equivalence of code sequences.

- The TOAST system is able to superoptimise 32-bit programs of up to five instructions long on modern machine architectures.
- The application of ASP to the problem domain demonstrates the ease of modelling machine architectures and the semantics of its instructions in the TOAST system.
- The relative performance of ASP tools within the TOAST system, with clear recommendations for the most efficient classes of grounders and solvers.

In Chapter 6 we apply the TOAST system to superoptimising sequences for the SPARC V8 architecture, while in Chapter 7 we utilise the whole of the TOAST system in generating equivalence classes of optimal sequences of length one upwards to generate a library of peephole optimisations for a specific machine architecture.

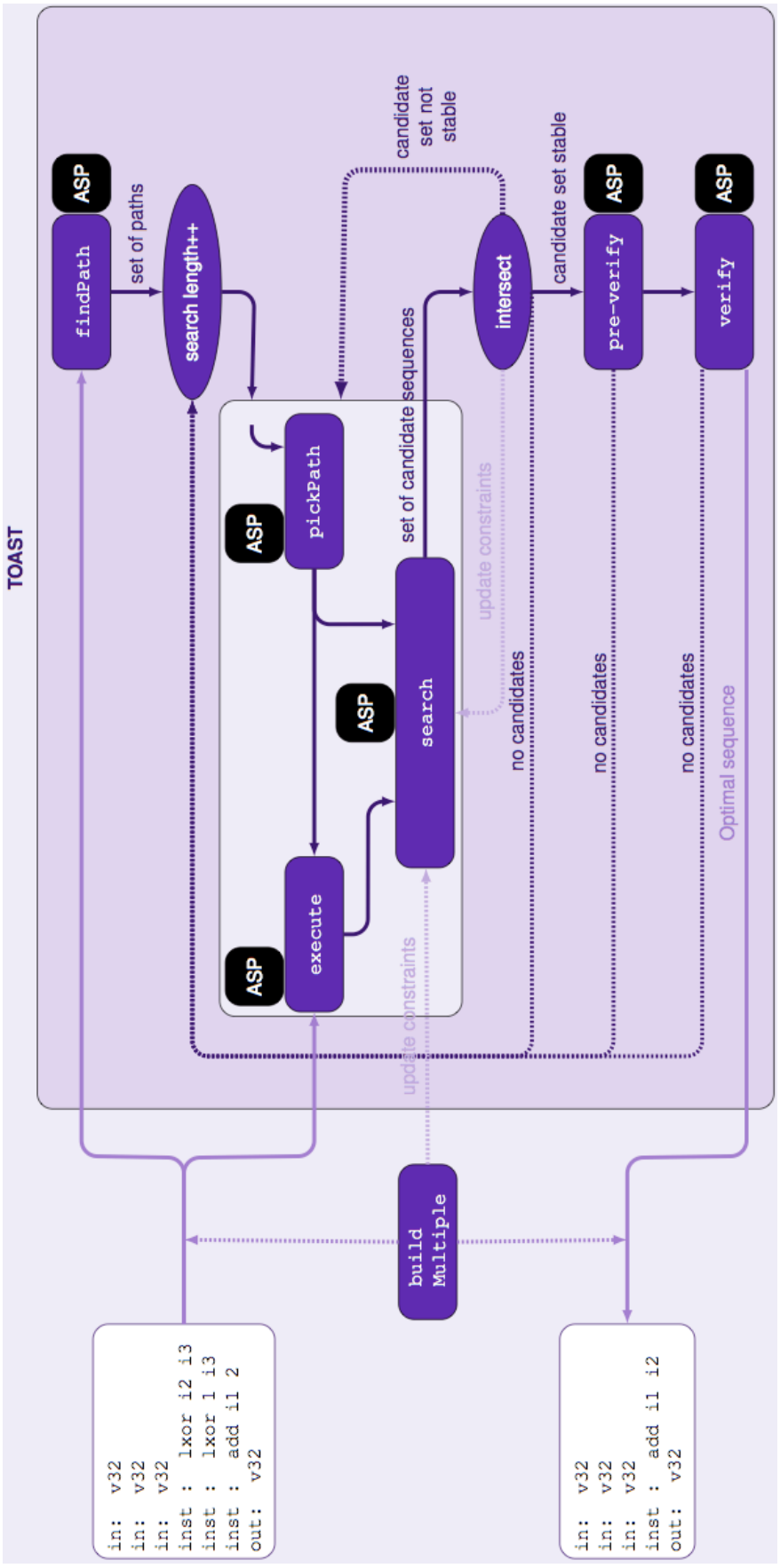

Figure 5-4: TOAST system architecture for an example superoptimisation process

Chapter 6

A Case Study: Superoptimising SPARC V8

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.

Richard Feynman

6.1 Introduction

In this chapter we apply the TOAST system (presented in Chapter 5) to superoptimising sequences for the SPARC V8 architecture. We provide benchmarks for a wide range of tests, demonstrating the interconnection of the system components, while also providing insight into the potential bottlenecks. We also compare aspects of the system to other existing superoptimising implementations.

6.2 Architecture Overview: SPARC V7/V8

SPARC (from *Scalable Processor Architecture*) is a load/store RISC architecture developed by Sun Microsystems in 1986. The SPARC V7 [12] and V8 [166] are microprocessor specifications and not specific implementations. They consist of a linear 32-bit address space with few and simple instruction formats; all instructions are 32 bits wide, and are aligned on 32-bit boundaries in memory. There are only three basic

instruction formats, and they feature uniform placement of opcode and register address fields. Only load and store instructions access memory and I/O. There are few addressing modes - a memory address is given by either register-register or register-immediate. The SPARC has triadic register addresses: most instructions operate on two register operands (or one register and a constant), and place the result in a third register [170]. The V7 uses 80-bit extended addressing (128-bit aligned) instead of quad word. The SPARC V8 differs from the V7 by the inclusion of improved multiply/divide instructions and tagged instructions; the tagged add/subtract instructions assume that the two least-significant bits of the operands are tag bits. A recent update to the SPARC architecture (V9) extends the addresses to 64-bit, including the addition of a number of instructions [180].

The SPARC V7/V8 architectures were chosen as a test architecture for the TOAST system due to the more complex modelling requirements compared to the MIPS R2000 architecture. While the SPARC architecture also represents a clean RISC model, it includes 64-bit extensions and more modern design features, such as tagged instructions. It was also a class of machine architecture that we have physical access to, allowing possible real system testing. In this way, it was chosen as a effective second validation architecture for the TOAST system.

The SPARC V7 and V8 architectures are used in both this chapter and also in Chapter 7; the architectural description for the both architectures can be found in Appendix C (page 117).

6.3 Superoptimising SPARC

To demonstrate the utility of the TOAST system we will use the following benchmark tests:

Search: `sequence5` (see page 55) generates programs that search the space of SPARC V8 instructions for candidate sequences for a program of five instructions. This sequence was selected as a worst-case, an example of a sequence that is already optimal, giving an approximate ceiling on the performance of the system.

Verify: consists of two tests: `verifytest1` which tests the non-trivial equivalence of two short code sequences, adding an unsigned number to itself and multiplying it by two; and `verifytest2`, which tests the non-equivalence of two code sequences, that only differ on one set of inputs and hence will result in the solver

returning only one answer set (which is an *AnsProlog* encoding of the inputs on which they differ).

TOAST: returning complete runs for the TOAST system, utilising all components (as presented in Chapter 5). Two tests: `argredundance`, which tests to see if a non-trivial redundant argument is optimised away, reducing a three instruction sequence to one instruction; and `signum`, which returns the sign of a binary integer, or zero if the input is zero. This second test is used to give a comparison to the GSO system [79] as presented in Chapter 3.

6.3.1 Searching

The `sequence5` test program, as given in Listing 6.1, is a test sequence that is already optimal, so it gives an approximate ceiling on the performance of the system in searching over the space of SPARC V8 instructions. Timings are given in Table 6.1 and plotted in Figure 6-1; solver timeouts occurred after 240 hours.

```
! input 1 in %i1
! input 2 in %i2
andcc %i1 %i2 %l1
addcc %i1 %l1 %l2
addcc %i1 %l2 %l3
addcc %i1 %l3 %l4
subcc %i0 %l4 %o1
! output in %o1
```

Listing 6.1: `sequence5` search test for SPARC V8

	Search sequence length				
Solver	**1**	**2**	**3**	**4**	**5**
clasp-1.2.1	0.28	2.01	189.00	5211.84	20625.16
cmodels-3.79	0.37	6.89	1019.00	4314.38	21699.27
smodels-2.33	0.28	7.57	6100.36	t/o	t/o
smodels-ie-1.0.0	0.21	6.91	2279.31	t/o	t/o
sup-0.4	0.44	3.15	177.24	5596.71	23012.61

Table 6.1: Timings (in sec) for `sequence5` search tests on SPARC V8

6.3.2 Verifying

The `verifytest1` program tests the (non-trivial) equivalence of two code sequences, as presented in Listing 6.2. The `verifytest2` program tests the non-equivalence

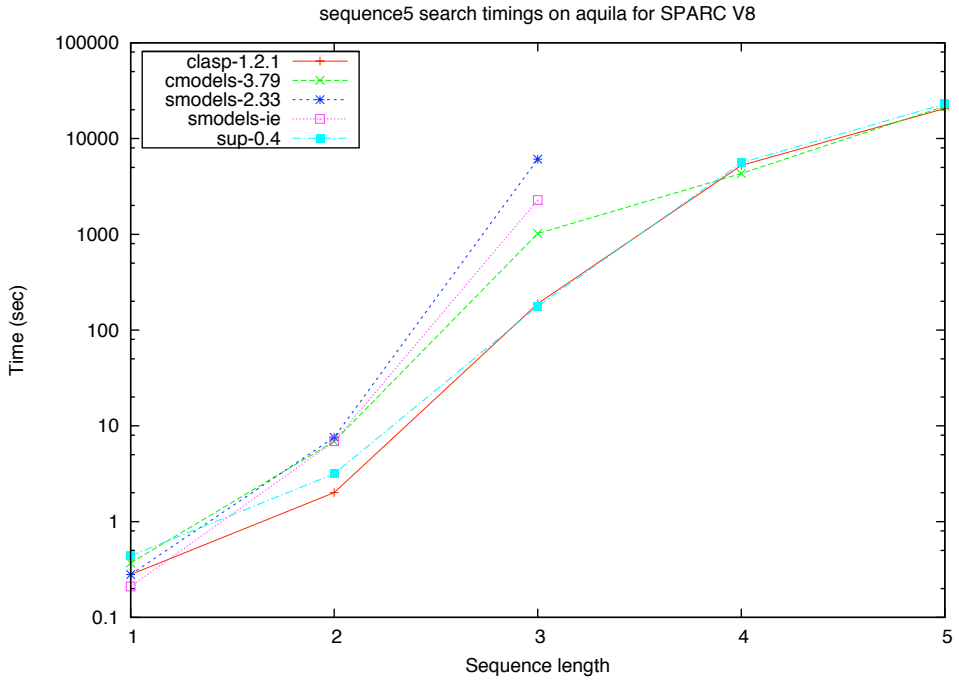

Figure 6-1: Plot of `sequence5` search test times (in sec) for increasing sequence length on SPARC V8

of two code sequences (as given in Listing 6.3), that only differ on one set of inputs. Timings for both tests for the five solvers are given in Table 6.2; again, solver timeouts occurred after 240 hours.

```
in: v32
in: i2
inst: multu i1 i2
out: v32

in: v32
in: i2
inst: add i1 i1
out: v32
```

Listing 6.2: `verifytest1` test programs for SPARC V8

```
in: v32
in: i1
in: i31
inst: srs32 i1 i3
inst: lorcc 1 1
inst: cbne 2
inst: add 1 i2
out: v32

in: v32
in: i1
in: i31
inst: addcc i1 i1
inst: csubcc i1 1
inst: cadd 2 i1
out: v32
```

Listing 6.3: `verifytest2` test programs for SPARC V8

	Verify tests	
Solver	`verifytest1`	`verifytest2`
clasp-1.2.1	15.20	8.17
cmodels-3.79	22.19	10.22
smodels-2.33	t/o	t/o
smodels-ie-1.0.0	t/o	t/o
sup-0.4	t/o	8.77

Table 6.2: Timings (in sec) for 32-bit verify tests on SPARC V8

6.3.3 TOAST System Benchmarking

For the TOAST system tests, we performed a full run through the TOAST process, utilising each component and validating the input and output we obtain. This would be performed automatically during a real system run, but has been decomposed to demonstrate the modular components of the system and how they interact.

```
in: v32
in: v32
in: v32
inst : xor i2 i3
inst : xor 1 i3
inst : add i1 2
out: v32
```

Listing 6.4: `argredundancetest` verify test for SPARC V8

Example 6.1. Using the `argredundance` test program as shown in Listing 6.4, we will demonstrate the control flow of the TOAST system:

Find all paths through the input program: this results in a direct instruction path of $1, 2, 3$, which means instruction one, two and three are executed sequentially. This is more relevant when there are multiple paths through a program to ensure that we test all possible paths.

Generate vectors and constraints for search space 1, run 1: we pick a path through the program using a set of selected vectors, to give each input a starting value. The instruction sequence is then executed using these chosen vectors to generate constraints, essentially giving us input and output values encoded in *AnsProlog* to perform the initial search.

Initial search over space 1: initial search over sequences of length one using the vectors and constraints. This generates the first results set of potential candidates. In this example we generate 25 candidates in 10 sec.

Generate vectors and constraints for search space 1, run 2: we pick a new set of vectors (input values) which enables us to generate a new set of constraints (output values). We then perform a new directed search over space 1.

Cut and search over space 1: we generate further search constraints (as shown in Listing 6.7) to trim the search space using information from the initial search and then re-search. We again generate 25 candidates in 11 sec. The criteria for searching means we always search at least twice to enable us to generate cut constraints; if the set of candidates varies (either increases or decreases) we continue searching, if it is empty we move onto the next space, if it stabilises we stop searching. In this example, we have generated 25 candidates on two distinct searches, so the set of candidates is stable and we move onto verification.

25 candidate(s) to verify: we perform the pre-verify heuristic on the 25 candidates (given in Listing 6.6). This is the representative test using a chosen set of input vectors to remove trivially incorrect sequences that have not been pruned during the search. This heuristic takes less than one second and discards 24 of the 25 sequences, with only `add i1 i2` (from line 22 in Listing 6.6) passing the test.

Full verify for candidate: add i1 i2: the full verify test checks to see if two sequences are equivalent over all input values, demonstrating full equivalence. In this case, the candidate passes a full verify in 1 sec and we have therefore found an optimal sequence of length one.

Summary: this full run of the TOAST system took 41 secs to complete, superoptimising the original length three sequence to a sequence of length one.

```
1   in: v32
2   in: i1
3   inst: subcc i1 i0
4   inst: cbg 4
5   inst: cbl 5
6   inst: lor i0 i0
7   inst: cba 4
8   inst: lor i0 i2
9   inst: cba 2
10  inst: sub i0 i2
11  out: v32
```

Listing 6.5: `signum` test program for SPARC V8

```
1   land i2 i2
2   sdiv i2 i0
3   lor i0 i3
4   add i3 i3
5   add i1 i1
6   add i2 i3
7   add i0 i1
8   lor i3 i3
9   add i1 i3
10  add i0 i0
11  lor i1 i2
12  lor i2 i2
13  sdiv i2 i3
14  lor i0 i1
15  umult i2 i2
16  lor i0 i0
17  lor i1 i1
18  smult i2 i2
19  add i0 i2
20  lor i1 i3
21  lor i2 i3
22  add i1 i2
23  add i0 i3
24  sdiv i2 i1
25  lor i0 i2
```

Listing 6.6: Superoptimised candidates generated from `argredundance` test on SPARC V8

A similar run to above was performed for the `signum` function as first introduced in Chapter 3 (page 16) and given in Listing 6.5. This is one of the sequences found by both Massalin [131] and GSO [80]. As expected, no candidates were found at sequence length one and two, whereas over 120 sequences were found on the initial search at sequence length three. This candidate set was pruned on the second search over length three and 13 sequences were passed to verify. All of these sequences passed the pre-verify stage and a full verify, confirming the sequences found by GSO. The total

```
1   :- not validSequence.
2   validSequence :- istream(blue,1,umult,i2,i2,none),  requireValidSequence.
3   validSequence :- istream(blue,1,smult,i2,i2,none),  requireValidSequence.
4   validSequence :- istream(blue,1,land,i2,i2,none),  requireValidSequence.
5   validSequence :- istream(blue,1,add,i0,i0,none),  requireValidSequence.
6   validSequence :- istream(blue,1,add,i0,i1,none),  requireValidSequence.
7   validSequence :- istream(blue,1,add,i0,i2,none),  requireValidSequence.
8   validSequence :- istream(blue,1,add,i0,i3,none),  requireValidSequence.
9   validSequence :- istream(blue,1,add,i1,i1,none),  requireValidSequence.
10  validSequence :- istream(blue,1,add,i1,i2,none),  requireValidSequence.
11  validSequence :- istream(blue,1,add,i1,i3,none),  requireValidSequence.
12  validSequence :- istream(blue,1,add,i2,i3,none),  requireValidSequence.
13  validSequence :- istream(blue,1,add,i3,i3,none),  requireValidSequence.
14  validSequence :- istream(blue,1,lor,i0,i0,none),  requireValidSequence.
15  validSequence :- istream(blue,1,lor,i0,i1,none),  requireValidSequence.
16  validSequence :- istream(blue,1,lor,i0,i2,none),  requireValidSequence.
17  validSequence :- istream(blue,1,lor,i0,i3,none),  requireValidSequence.
18  validSequence :- istream(blue,1,lor,i1,i1,none),  requireValidSequence.
19  validSequence :- istream(blue,1,lor,i1,i2,none),  requireValidSequence.
20  validSequence :- istream(blue,1,lor,i1,i3,none),  requireValidSequence.
21  validSequence :- istream(blue,1,lor,i2,i2,none),  requireValidSequence.
22  validSequence :- istream(blue,1,lor,i2,i3,none),  requireValidSequence.
23  validSequence :- istream(blue,1,lor,i3,i3,none),  requireValidSequence.
24  validSequence :- istream(blue,1,sdiv,i2,i1,none),  requireValidSequence.
25  validSequence :- istream(blue,1,sdiv,i2,i0,none),  requireValidSequence.
26  validSequence :- istream(blue,1,sdiv,i2,i3,none),  requireValidSequence.
27  requireValidSequence.
```

Listing 6.7: Constraints generated by `searchCut` by superoptimising `argredundance` verify test on SPARC V8

runtime for this test was 144 seconds, in comparison to a GSO runtime of under 10 seconds. Although the results for the TOAST system are not strictly competitive with respect to time, it is important to note that the results are validated as correct by both superoptimising implementations. This will be discussed in more detail in Section 6.4.

6.4 Discussion

As presented in the previous section, we have further reinforced the applicability of the TOAST system in superoptimising sequences for 32-bit machine architectures. A high-level comparison between the results for searching and verifying sequences for the MIPS R2000 and SPARC V8 architecture indicates that the complexity of the modelling instruction set architecture has a small effect on the benchmark results. Even though the number of instructions provided by an architecture is a crude metric of complexity, the SPARC V8 architecture provides more than twice as many instruction as the MIPS R2000 architecture. However, many of these are undefined within the TOAST model due to the lack of floating point operations on the MIPS R2000, but the SPARC V8 is a more modern RISC architecture with more complex instructions, such as tagged arithmetic. There also exists significant complexity issues with the

SPARC instruction pipeline and how this complicates generating and optimising code for that machine architecture, especially with respect to instruction scheduling and data dependencies [166]. The values for searching up to length five are approximately the same, with the discrepancy attributed to the wider range of instructions available on the SPARC V8. The verify tests are different for the two architectures, but general comparisons for solving 32-bit programs indicates that again this is achievable for the TOAST system. The two types of verify test demonstrates that not only can TOAST verify non-trivial programs, but it can also quickly verify when sequences are not equivalent, and on which inputs they differ. Ideally, the search heuristics and the pre-verify step would prune all non-equivalent sequences, but it is feasible that a sequence could reach a full verify and fail. This is an important validation of the interaction and functionality of the various components of the TOAST system, along with highlighting the importance of the full verify stage.

The results presented in this chapter indicate that while TOAST can confirm and validate results of existing superoptimising implementations, it is currently not as competitive from a runtime perspective in certain areas, especially against the goal function approach of GSO. However, a direct comparison is not necessarily a fair one: goal-directed functions (especially written in C or assembly language) will generate certain candidate sequences very quickly due to precise and efficient encoding of that specific problem description. TOAST is able to accept any arbitrary instruction sequence as input, whether that describes the specific goal used in the `signum` test, or a slight variation. GSO, for example, would not be able to handle this variation, as it would require defining a new specific goal function and how this maps to each machine architecture. For TOAST, this change is trivial; this flexibility is an important difference between the systems and may offset certain performance considerations. The TOAST system generated more sequences during the initial search phase for the `signum` test, which indicates that improvements could be made in the search heuristics. However, the verification step was a full equivalence test for all input values and performed for all of the equivalent sequences.

The results presented in this chapter indicate it would be possible to harvest instruction sequences from a suitable SPARC V7/V8 binary program source and attempt to optimise sequences of up to length five. This idea will be discussed further in Chapter 8.

6.5 Summary

In this chapter we have used the TOAST system to superoptimise sequences for the SPARC V8 architecture. We have shown that:

- The TOAST system is able to superoptimise real code sequences for a complex 32-bit machine architecture.
- In comparison to existing superoptimising implementations, the TOAST system produces results that are guaranteed optimal: no further manual checking step is required.
- The flexibility of the TOAST system for accepting arbitrary sequences as input is more applicable to potential future application areas than restricted goal-directed superoptimisers.

In the following chapter, we present one of these possible application areas for the TOAST system: generating all optimal sequences of length one, which are then used to generate all optimal sequences of length two and so on, to construct a library of equivalence classes of instruction sequences that can be used in a peephole optimiser. This library of optimal sequences also has potential application into the optimisation phases of a standard compiler toolchain.

Chapter 7

buildMultiple: A Peephole Superoptimiser

The lurking suspicion that something could be simplified is the world's richest source of rewarding challenges.

Edsger W. Dijkstra

7.1 Introduction

As first introduced in Chapter 2, peephole optimisation is a technique for locally improving code sequences by substituting shorter or faster sequences in a small window, known as the "peephole" [40, 133]. It is characteristic of peephole optimisation that each improvement may spawn opportunities for additional improvements, such as redundant instruction elimination or algebraic simplifications [2, 41]. Most modern compiler toolchains utilise forms of peephole optimisation, whether used during the construction of the intermediate representations [172], or as a post-code generation optimisation phase [41].

In this chapter, we present an application of the TOAST system: a peephole superoptimiser based upon a generated library of all optimal sequences for a specific machine architecture. We present the rationale for this approach to generating equivalence classes of optimal sequences, describe how we utilise the TOAST system and give experimental results for the SPARC V7 architecture.

7.2 Motivation

Peephole optimisers have found widespread use in most modern compiler toolchains, with the technique first identified by McKeeman in the 1960s [133]. There has been a rich history of developing and applying peepholing techniques [40, 41, 172], for example using architectural descriptions [102], combining with register allocation [43] and even using superoptimising techniques [14]. There have also been declarative approaches to generating rules for peephole optimisation using a form of string pattern matching [167].

The *buildMultiple* tool utilises the components of the TOAST system to build and refine a set of constraints which augment the search component. Its design is based on the observation that an optimal sequence of instructions will not contain any sub-optimal instruction sequence. As was demonstrated in the previous chapters, the TOAST system can easily perform directed searches for sequences that meet specified criteria; these sequences are then verified for equivalence. The flexibility of ASP allows the addition of constraints to enable the generation of large search programs that generate all optimal sequences of a given length for a specific number of inputs.

By generating this library of optimal sequences for a given instruction length and number of inputs, it is possible to apply this information to optimising any code sequence for the chosen architecture. While this process may take a significant amount of time to generate these sequences (potentially of the order of months of compute time), this would only ever need to be performed once per architecture model. In this way, the high up-front computational cost is mitigated by its long-term use.

7.3 System Components

The search component of TOAST is used to generate the set of all possible instructions sequences for the tuple of instruction length and number of inputs. This search set is then superoptimised using the TOAST system; if they are found to be sub-optimal or equivalent to an existing optimal sequence then they are abstracted away to form additional constraints for the run. If they are found to be optimal they are marked as such and output. If anything shorter in instruction length than the current sequence is found, it is clearly non-optimal; if it is found to only optimise to itself, it is again marked as an optimal sequence; if it fuzzy matches to itself (in which orderings of inputs are taken into consideration) then these multiple orderings are equivalent and they need to be saved. If it matches another candidate then the two are marked as

equivalent and the other sequence is removed. If it fuzzy matches another target then all of the re-orderings of that are equivalent to the same re-orderings of the target - thus the other sequence is removed. A summary of the actions for handling sequences in *buildMultiple* are presented in Table 7.1.

Result	**Action**
Shorter than target	Mark target as non-optimal
Equal to target	Ignore
Fuzzy match to target	Remove reorderings and keep
Equal to other target	Remove other
Fuzzy match to other target	Remove other
Other	Mark as non-optimal

Table 7.1: *buildMultiple* action overview

Due to the exponential nature of the *buildMultiple* process, the use of search heuristics is vital to prune the large search spaces as much as possible. A large number of candidates (hundreds for length one) will be generated during the search, which will then need to be verified to identify equivalences and which sequences can be discarded. Although this procedure is time consuming and computationally expensive, it produces very strong sets of constraints (in *AnsProlog*) and only ever needs to be run once for a given architecture. Even if this takes a number of months to complete, the resulting equivalence classes generated would provide an ample source of optimal code sequences for use within a peephole optimiser. It also highlights a key advantage of using *AnsProlog*; the flexibility to easily add extra constraints without altering the search algorithm. With a procedural system (like all previous superoptimising implementations) *buildMultiple* would simply not be feasible.

7.4 A *buildMultiple* Library for SPARC V7

In this section, we present the results for using *buildMultiple* to generate equivalence classes of optimal sequences for the SPARC V7 architecture. We show timings and sequence statistics for instruction sequence lengths one to four, with number of inputs from one to six (dependent on instruction length).

The SPARC V7 architecture has 194 instructions defined in the architecture description [12]: 23 instructions declared for search (which means that the internal single instruction form should be included in both the search and the execute space), 17 instructions declared for the execute space only and six declared as combo instructions

(which means there must exist a single instruction definition for search or exec elsewhere). The represents the raw space of instruction combinations, which are pruned and optimised to make searching and verifying sequences feasible.

Due to the large number of results generated for sequences of instruction length two and higher, we have not been able to present the results in the body of this thesis; however, these are available from the author on request. An overview of the *buildMultiple* run results is given in Table 7.2, with timings plotted in Figure 7-1. A plot of the relationship between the number of optimal and non-optimal sequences generated in each run is given in Figure 7-2. *buildMultiple* sequence information for the one instruction-one input run is given in Table 7.3; for the one instruction-two input run in Table 7.4.

Instructions	**Inputs**	**Non-Optimal**	**Optimal**	**Run Time (secs)**
1	1	0	4	23
1	2	0	3	31
1	3	0	0	<1
1	4	0	0	<1
2	1	83	34	234
2	2	98	190	906
2	3	32	74	498
2	4	0	0	2
2	5	0	0	2
3	1	135	12	812
3	2	201	52	3276
3	3	88	220	7762
3	4	41	98	5919
3	5	0	0	7
3	6	0	0	8
4	1	203	3	32805
4	2	348	23	99124
4	3	525	102	203564
4	4	297	326	356374
4	5	122	187	181378
4	6	46	83	69202

Table 7.2: *buildMultiple* sequence statistics for lengths 1–4 on SPARC V7

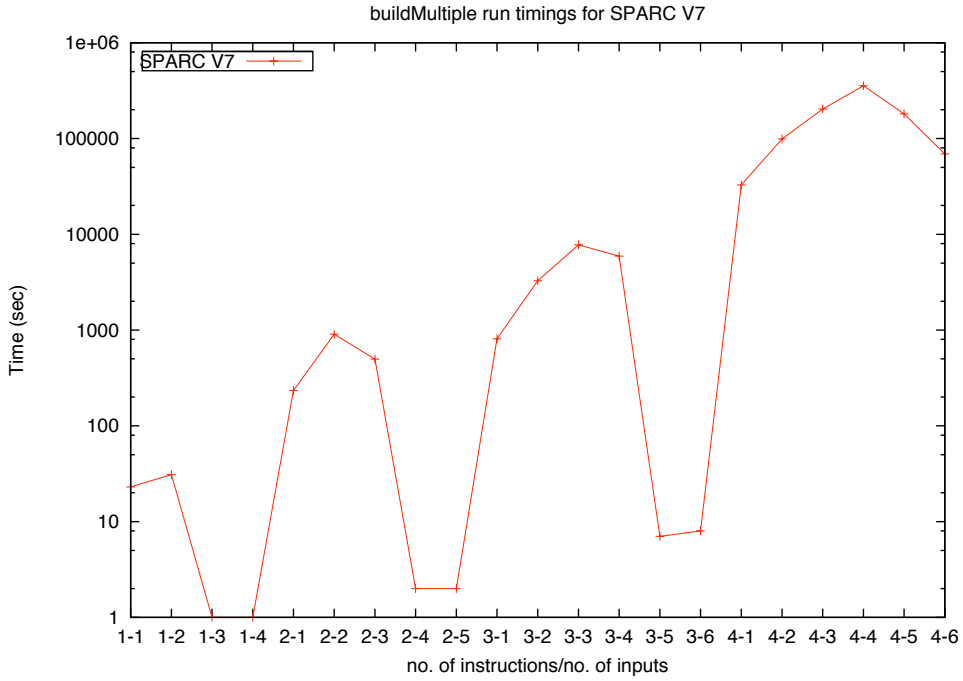

Figure 7-1: Plot of *buildMultiple* run timings (in sec) for SPARC V7

7.5 Discussion

There are some interesting results presented from applying *buildMultiple* to the SPARC V7 architecture. While it is a computationally expensive task, it is a worthwhile process to generate all optimal sequences of a given length and number of inputs. As expected, the timings shown in Figure 7-1 indicate the exponential nature of the problem, although it is somewhat mitigated by the method in which we generate the sequences. By using the constraints created by generating length one sequences, we can use these to prune the search space for generating all sequences of length two and so on. A problem with this approach is the sequential nature the task; parallelisation is possible, but this means that the significant benefit of using the constraints generated by the previous instruction length to prune the search space is lost.

The plot in Figure 7-2 that shows the number of optimal versus non-optimal sequences generated on each run is fairly intuitive and as expected. For example, for the results produced from the one instruction-one input run, the number of optimal sequences generated is four; this does not indicate four sequences, but four distinct equivalence

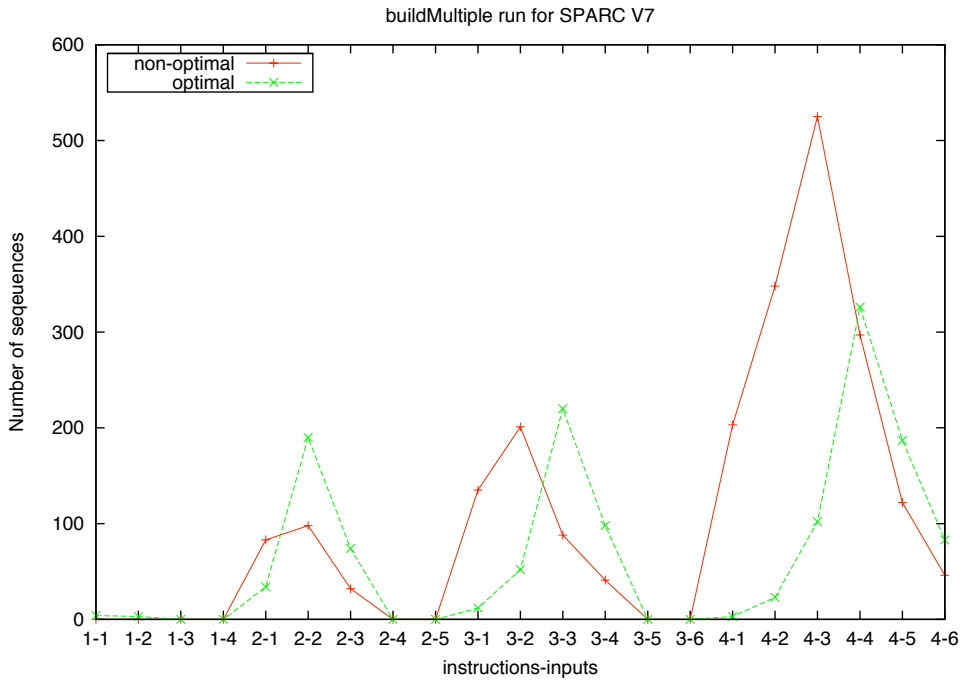

Figure 7-2: Plot of optimal/non-optimal sequences generated by *buildMultiple* run for SPARC V7

classes of unique optimal sequences.

Even though a significant number of sequences are marked equivalent and kept, or are equivalent and removed, the key relationship is between the number of sequences marked optimal and non-optimal. As can be seen in Figure 7-2, both the number of optimal and non-optimal sequences increase exponentially, but the number of optimal sequences increases more slowly. This is what we would expect of a well-designed instruction set architecture: if a large number of optimal sequences were found as the instruction lengths increase, this would imply something about the available instructions of the architecture, if they easily combine to form optimal sequences. If the converse were true, if a large number of non-optimal sequences were found as the instruction length increased, this may imply something negative about the design of the instruction set, with poorly selected instructions. This is a interesting general point from the results generated by *buildMultiple*, whether it is possible to infer information about the design of the instruction set architecture and the measure of orthogonality of the instructions. The data for the SPARC V7 architecture implies that there exists redundancy in the instruction set. This may allow testing of instruction sets from a specific

Candidate	Equivalent

`inst: adcc i1 i1`	`inst: add i1 i1`
	`inst: addcc i1 i1`
	`inst: taddcc i1 i1`

`inst: lnot i1`	`inst: lnot i1`
`inst: srs32 i1 i1`	`inst: srs32 i1 i1`

`inst: lor i1 i1`	`inst: lorcc i1 i1`
	`inst: lor i1 i1`
	`inst: land i1 i1`
	`inst: landcc i1 i1`

`inst: subcc i1 i1`	`inst: lxor i1 i1`
	`inst: subcc i1 i1`
	`inst: sub i1 i1`
	`inst: lxorcc i1 i1`
	`inst: tsubcc i1 i1`

`inst: slz32 i1 i1`	`inst: slz32 i1 i1`
`inst: srz32 i1 i1`	`inst: srz32 i1 i1`

Summary

`addcc i1 i1`	equivalent-keep
`lnot i1`	optimal
`srs32 i1 i1`	optimal
`lor i1 i1`	equivalent-keep
`lorcc i1 i1`	equivalent-remove
`subcc i1 i1`	equivalent-keep
`land i1 i1`	equivalent-remove
`taddcc i1 i1`	equivalent-remove
`sub i1 i1`	equivalent-remove
`slz32 i1 i1`	optimal
`srz32 i1 i1`	optimal
`add i1 i1`	equivalent-remove
`lxor i1 i1`	equivalent-remove
`landcc i1 i1`	equivalent-remove
`lxorcc i1 i1`	equivalent-remove
`tsubcc i1 i1`	equivalent-remove

Table 7.3: *buildMultiple* generated equivalent sequences for one instruction-one input on SPARC V7

theoretical perspective, assuming that the single instructions are a baseline and look at how combinations of instructions interact and what functionality they provide. This level of meta-analysis is problematic, as it may be possible to infer too much from the

Candidate	Equivalent
`inst: lxorcc i1 i2`	`inst: lxor i1 i2` `inst: lxorcc i1 i2`
`inst: srs32 i1 i2`	`inst: srs32 i1 i2`

`inst: addcc i1 i2`	`inst: addcc i1 i2`
	`inst: add i1 i2`
	`inst: taddcc i1 i2`

`inst: landcc i1 i2`	`inst: landcc i1 i2` `inst: land i1 i2`

`inst: tsubcc i1 i2`	`inst: subcc i1 i2`
	`inst: sub i1 i2`
	`inst: tsubcc i1 i2`

`inst: slz32 i1 i2`	`inst: slz32 i1 i2`
`inst: srz32 i1 i2`	`inst: srz32 i1 i2`
`inst: lorcc i1 i2`	`inst: lor i1 i2` `inst: lorcc i1 i2`

Summary

`lxorcc i1 i2`	equivalent-keep
`srs32 i1 i2`	optimal
`addcc i1 i2`	equivalent-keep
`landcc i1 i2`	equivalent-keep
`taddcc i1 i2`	equivalent-remove
`tsubcc i1 i2`	equivalent-keep
`lxor i1 i2`	equivalent-remove
`land i1 i2`	equivalent-remove
`slz32 i1 i2`	optimal
`add i1 i2`	equivalent-remove
`subcc i1 i2`	equivalent-remove
`srz32 i1 i2`	optimal
`lorcc i1 i2`	equivalent-keep
`lor i1 i2`	equivalent-remove
`sub i1 i2`	equivalent-remove

Table 7.4: *buildMultiple* generated equivalent sequences for one instruction-two inputs on SPARC V7

results, more so since it would be prudent to obtain further data points to see how the timings scale for increasing instruction length and number of inputs. Furthermore, the measure of optimality of an instruction set is certainly a metric, but there is a balance between providing an optimised instruction set that has esoteric and complex instruc-

tion semantics, if the users or developers are unable to understand or use it efficiently: there is a balance between optimality, maintainability and usability [147].

A phenomenon that is apparent from Figure 7-2 is that the number of sequences increases steadily, but decreases when the number of inputs hits a limit, and then drops to zero. This can be explained by analysing the maximum number of inputs it is feasible to use in a sequence of length n: this is $2n$, as it is only possible to utilise at most two inputs per instruction, due to the triadic addressing in the test architectures. Therefore, the decline to zero is explained by reaching this limit; for example, for two instructions-four inputs; for three instructions you would expect this to occur at six inputs, but in this case occurs at five. Reasons for this disparity may be explained by a proportion of the generated combinations consisting of instructions that only require one input. The converse case of fixing the number of inputs and constraining the number of instructions is not true: the instruction length can increase arbitrarily as you could be performing some mathematical operation that utilises say two inputs and performs some sort of unrolled iteration or summation with a large number of instructions.

Another scenario is that as the instruction length increases, *buildMultiple* may generate no optimal sequences for a certain number of inputs, but then by introducing another input generates a whole set of optimal sequences. The relationship between number of instructions and inputs is complex and has an strong effect on the number of sequences generated and the time taken to find them. Further research and testing is required to infer more about this relationship.

The question of how do we verify that the generated sequences are valid and correct, is demonstrated by referencing back to the architectural model and then hand-checking a subset of sequences to ensure they are correct. However, the design of the system is such that the constraints we include decide which sequences are generated. One issue is the ramifications of problems in an earlier run; if a mistake is made for a length one sequence that is marked as optimal and it is non-optimal, then this would have an effect on the constraints for generating length two sequences and so on. This scenario is mitigated by verification of the search and the heuristics used within the TOAST system. However, as with the TOAST system, the optimality of the sequences generated is with respect to the architectural model.

However, it is also possible that the numbers generated from a *buildMultiple* run do not allow us to draw any significant conclusions, but just confirms intuition about what would be expected for a well-designed microprocessor architecture. The results may provide some insight into the upper bounds of searching and verifying with the

TOAST system, but the emphasis is on the methodology and whether it is scalable. Results for sequences of length five would provide an interesting data point, as this would provide insight into how the timings scale. With sequences of length four taking approximately four to five days, a sensible estimation on the time taken to generate length five sequences would be of the order of four to five months, extrapolating from the existing data.

As mentioned previously, the intended application of these generated equivalence classes is as a library of optimal sequences for use as a peephole optimiser (as introduced in Chapter 2 (page 11). It could be applied to optimising object code or utilised in the code optimisation phase of a compiler toolchain (similar to how GSO contributed to GCC [79]). The canonicalisation of sequences is important to ensure they are abstracted and can be applied generically, along with efficient encoding, storage and retrieval of the sequences.

The main peepholing implementations [40, 42, 43, 133, 167] rely on significant code analysis and pattern matching, generating sequences from machine descriptions. With *buildMultiple*, we work from the machine description and the available instructions to generate all optimal sequences of length one upwards, for a given number of inputs, which are then used to generate sequences of length two, and so on. This approach is different and initially computationally more expensive than approaches for existing peephole optimisers, as we generate equivalence classes of optimal sequences rather than directly looking for basic pre-encoded program transformations.

The Stanford superoptimiser [14] performs automatic generation of peephole optimisations using a brute force superoptimising approach, with some successful results presented for the Intel x86 architecture. Their approach relied on harvesting sequences from a large repository of pre-existing Intel x86 binary programs, to automatically generate optimisations. A similar approach has also been applied to binary translation [15].

7.6 Summary

In this chapter, we have presented a significant application of the TOAST system as a peephole superoptimiser, based on the creation of a library of equivalence classes of optimal sequences. We have demonstrated the technique for using the TOAST system and the rationale for generating all optimal sequences of a certain length and number of inputs and then identifying equivalences between these sequences. We showed the

viability of the *buildMultiple* approach by generating all optimal sequence of instruction length one to four with varying number of inputs on the SPARC V7 architecture, with a discussion of what the *buildMultiple* results implies about the design of the instruction set architecture, how this technique can scale for longer sequences and how it can be applied in the future. A further discussion of this future work and potential applications for *buildMultiple* is presented in Chapter 8.

Chapter 8

Concluding Remarks

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

Optimisation in modern compilers is an accepted misnomer for *performance improvement some of the time*. With the emergence of resource-critical environments, such as the embedded domain, new optimisation strategies are required. New metrics of optimality, such as low memory usage and low power consumption will be of huge importance. However, due to the potentially conflicting nature of these metrics, we may soon encounter constrained optimisation problems, for example, attempting to optimise for both program size and low power consumption. This is an established problem for modern compiler toolchains, such as GCC, and needs to be considered when optimising for multiple metrics in the TOAST system.

Superoptimisation is one possible approach to the code optimisation problem. Due to the exhaustive nature of the approach, superoptimisation has previously not scaled for optimising significant real-world code sequences. Nevertheless, a true code optimiser is not possible at present (if even theoretically possible due to the lack of program and data analysis); for example, would this true optimiser recognise naive code for a bubble sort and replace this with a quicksort algorithm? This would require significant static and runtime analysis, along with an estimation of the likely input data. In the practical case, this may not be the panacea of optimisation. Nevertheless, the TOAST approach to superoptimisation is an important step to providing a structured framework for generating truly optimal code sequences for a given microprocessor architecture.

The TOAST approach is a significant research step in developing superoptimisation as a practical technique and applying it to mainstream code optimisation. The TOAST system is able to superoptimise 32-bit code sequences of up to five instructions long, which could be further extended with new heuristics and improved searching and verification techniques. The peephole superoptimiser approach is a valid application for the TOAST system, with the generation of all optimal sequences of a certain length and number of inputs providing an significant resource for optimising machine code. The large up-front computation cost is mitigated by the need to only ever run once per machine architecture.

Issues with verification of the architectural model to the functionality of the physical device indicates that we are only ever able to optimise to the constrained model we have implemented, which may not fully encapsulate the functionality of the physical processor. The architectural model has been abstracted and TOAST has generated optimal code to the constraints of the model, but there are almost certainly improvements to be made to each model. Fortunately, the use of *AnsProlog* makes this improvement process simple, as is it possible to make amendments to any architecture or instruction description and re-run a test. The availability of efficient off-the-shelf solver tools, means that while we are reliant on external development resources, we can contribute to these communities and projects by pushing the state of the art and providing complex benchmark cases.

8.1 Major Contributions

The major contributions of this thesis are as follows:

1. Development of a practical and adaptable superoptimising code generation system based on ASP technology, presenting a structured approach to optimisation, with proof of optimality for acyclic code sequences. This functionality has been benchmarked and compared against existing superoptimising implementations.

2. A demonstration that superoptimisation of code is achievable in the general case and can be used to generate provably optimal code sequences for 32-bit architectures (and that extending the functionality for 64-bit architectures is also achievable).

3. Observations on the performances of a range of ASP solver tools, notably the performance of the more recent SAT-based and clause learning solvers compared

to the traditional backtracking solvers, on a complex real-world problem.

4. Demonstrating that ASP is an appropriate modelling paradigm for reasoning about large-scale, real-world problems. The application of ASP to the code optimisation problem has also contributed to the ASP community and will further stimulate future tool development.

8.2 Future Work

While some significant work has been achieved in this dissertation, there are numerous extensions and directions for future work:

New metrics of optimality: adapting the TOAST system to optimise for new metrics, such as low power consumption, runtime speed and memory usage. This would present a range of modelling problems, especially with regards to analysing and encapsulating the power consumption of specific instructions, but this is a logical extension of the instruction length optimality upon which the system is currently based. This may create linear optimisation and constrained optimisation problems whilst attempting to optimise for these new metrics or more than one metric in the optimisation model. It should be noted that optimising 64-bit architectures does not pose any new problems (and is in fact tractable with existing tools), it just requires more computational resources.

Optimising longer code sequences: if we are currently unable to optimise sequences of length eight, it should be feasible to break an eight instruction sequence into two tractable sub-sequences and optimise them. If we decompose the length eight sequence into two sub-sequences of length five and length three, or two sequences of length four, is it may be possible to combine the results of the superoptimised sub-sequences to get an optimised version of the length eight sequence. However, this will not necessarily be the optimal solution, only potentially an improved sequence. The benefit of breaking longer sequences into sub-sequences is that is then becomes possible to parallelise the system. There already exist aspects of the TOAST system which are inherently parallel, and recent work [24,27,54,149] on parallel ASP solving tools would further support a distributed approach. However, there still remains questions about the nature of local and global optimisations [2], especially whether it is more effective to optimise at a global level rather than focus on local peephole-like optimisations.

Nevertheless, this could be a promising approach to targeting longer sequences than the TOAST is currently able to optimise.

Targeting the embedded domain: this would be a significant domain to validate the TOAST system, by focusing on superoptimising a suitable embedded platform, such as the ARM microprocessor family [1], and developing research and development collaboration with industry. There would be significant new modelling challenges, for example, implementing predicated instructions (which enable conditional execution of instructions) and other DSP-like features of the ARM instruction set architecture [161], but this represents a key future development and target domain for the TOAST system.

Optimising for multi-core and multi-threaded architectures: at present the TOAST system has only modelled single-core RISC architectures. A next step would be to extend the system to optimise code for multi-core or multi-threaded architectures. Again, this presents significant modelling challenges, but it should be possible to extend the TOAST model to encapsulate this functionality and its semantics. With the development of microprocessor architectures favouring multi-core and multi-threaded designs [82], this is an important area to focus upon.

Integration into compiler toolchains: for example, the GNU Compiler Collection (GCC); this would be a significant metric of success for the TOAST system, if it was utilised for part of a standard compiler toolchain, or its output was integrated into its optimisation phases. This has already been achieved by an existing superoptimising implementation (GSO), so this indicates that there is still a contribution to be made. The use of the *buildMultiple* approach to generating equivalence classes of optimal sequences for a peephole library for a specific architecture could provide a rich source of information and may also allow us to analyse the design of instruction sets. This may also provide way of bounding what is possible via conventional optimisation, perhaps empirically evaluating to what level code is improved by modern optimising compilers and the theoretical limit to code optimisation.

General modelling: there exists a plethora of modelling problems that pose interesting questions, including extending the TOAST system to model conditional

[1] ARM announced in February 2009 that it had shipped ten billion ARM-powered processors to the mobile device market – the equivalent of 2.5 ARM processors for every mobile cellular subscriber in the world [10].

branches, unrolling loops, optimising in immediate values, perhaps even developing a strategy for modelling floating point operations. To enable future application of the TOAST system to optimising code for real-world platforms, some of these will have to be addressed.

Bibliography

[1] The Aggregate Magic Algorithms. `http://aggregate.org/MAGIC/`. [accessed 2009-08-01].

[2] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2nd edition, 2006.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[4] AMD, Inc. *AMD Athlon Processor x86 Code Optimization Guide*. AMD, Inc., 2002. Version 22007 Revision K.

[5] AMD, Inc. *Software Optimization Guide for AMD64 Processors*. AMD, Inc., 2005. Version 25112 Revision 3.06.

[6] Sean Eron Anderson. Bit Twiddling Hacks. `http://graphics.stanford.edu/~seander/bithacks.html`. [accessed 2009-08-01].

[7] Christian Anger, Martin Gebser, Thomas Linke, Andr Neumann, and Torsten Schaub. The NOMORE++ Approach to Answer Set Solving. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *LNCS*, pages 95–109. Springer, 2005.

[8] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2004.

[9] Krzysztof R. Apt and Roland Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19:9–71, 1994.

[10] ARM Holdings. ARM Announces 10 Billionth Mobile Processor. `http://www.arm.com/about/newsroom/24403.php`, February 2009. [accessed 2009-08-01].

[11] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 92(2):449–466, 2005. Special issue on Program Generation, Optimization, and Adaptation.

[12] Atmel Corporation. *SPARC V7 Instruction Set*, 2001. Revision 4168CAERO08/01.

[13] Marcello Balduccini and Michael Gelfond. Diagnostic Reasoning with *A-Prolog*. *Theory and Practice of Logic Programming*, 3(4):425–461, 2003.

[14] 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, 2006.

[15] Sorav Bansal and Alex Aiken. Binary Translation Using Peephole Superoptimizers. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[16] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[17] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19-20:73–148, 1994.

[18] Michael Beeler, William Gosper, and Richard Schroeppel. HAKMEM. A.I. Laboratory Memo 239, Massachusetts Institute of Technology, 1972.

[19] Jon Bentley. *Programming Pearls*. ACM Press, 2nd edition, 1999.

[20] David Bernstein, Michael Rodeh, and Izidor Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers*, 38(9):1308–1313, 1989.

[21] Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 160–174. Springer, 2008.

[22] Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. ANTON: Composing Logic and Logic Composing. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 542–547. Springer, 2009.

[23] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. An Application of Answer Set Programming: Superoptimisation - A Preliminary Report. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR'06)*, pages 258–266, 2006.

[24] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying Answer Set Programming to Superoptimisation. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, volume 4079 of *LNCS*, pages 270–284. Springer, 2006.

[25] Martin Brain and Marina De Vos. The Significance of Memory Costs in Answer Set Solver Implementation. *Journal of Logic and Computation*, 19(4):615–641, 2009.

[26] Joe Bungo. The Use of Compiler Optimizations for Embedded Systems Software. *Crossroads*, 15(1):8–15, 2008.

[27] F. Calimeri, S. Perria, and F. Ricca. Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms*, 63(1-3):34–54, 2008.

[28] Tran Cao Son, Enrico Pontelli, and Chiaki Sakama. Logic Programming for Multi-Agent Planning with Negotiation. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, pages 99–114. Springer, 2009.

[29] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEE Computer Society, 2007.

[30] Pawel Cholewinski, Victor W. Marek, and Mirosław Truszczyński. Default Reasoning System DERES. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 518–528, 1996.

[31] K. L. Clark. Negation as Failure. In *Logic and Data Bases*, pages 293–322, 1977.

[32] Jacques Cohen and Timothy J. Hickey. Parsing and Compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.

[33] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Systeme de Communication Homme-Machine en Francais. Technical report, Université de Aix-Marseille, France, 1973.

[34] Alain Colmerauer. Prolog in 10 Figures. *Communications of the ACM*, 28(12):1296–1310, 1985.

[35] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*, pages 37–52. ACM, 1993.

[36] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: Adaptive Compilation Made Efficient. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'05)*, pages 69–77. ACM, 2005.

[37] Tom Crick, Marina De Vos, Martin Brain, and John Fitch. Generating Optimal Code using Answer Set Programming. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 554–559. Springer, 2009.

[38] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[39] Shuhaizar Daud, R. Badlishah Ahmad, and Nukala S. Murthy. The Effects of Compiler Optimisations on Embedded System Power Consumption. *International Journal of Information and Communication Technology*, 2(1-2):73–82, 2009.

[40] Jack W. Davidson and Christopher W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, 1980.

[41] Jack W. Davidson and Christopher W. Fraser. Eliminating Redundant Object Code. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*, pages 128–132. ACM, 1982.

[42] Jack W. Davidson and Christopher W. Fraser. Automatic Generation of Peephole Optimizations. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 111–116. ACM, 1984.

[43] Jack W. Davidson and Christopher W. Fraser. Register Allocation and Exhaustive Peephole Optimization. *Software: Practice and Experience*, 14(9):857–865, 1984.

[44] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[45] Marina De Vos, Tom Crick, Julian Padget, Martin Brain, Owen Cliffe, and Jonathan Needham. LAIMA: A Multi-agent Platform Using Ordered Choice Logic Programming. In *Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT 2006)*, volume 3904 of *LNCS*, pages 72–88. Springer, 2006.

[46] Marina De Vos and Dirk Vermeir. Extending Answer Sets for Logic Programming Agents. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):103–139, 2004.

[47] James P. Delgrande, Torsten Grote, and Aaron Hunter. A General Approach to the Verification of Cryptographic Protocols Using Answer Set Programming. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 355–367. Springer, 2009.

[48] Marc Denecker. Whats in a Model? Epistemological Analysis of Logic Programming. In *Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR2004)*, pages 106–113. AAAI Press, 2004.

[49] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The Second Answer Set Programming Competition. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 637–654. Springer, 2009.

[50] Júrgen Dix, Ulrich Furbach, and Ilkka Niemelä. *Handbook of Automated Reasoning*, chapter Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations, pages 1241–1354. Elsevier, 2001.

[51] Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. In *Proceedings of 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 2919 of *LNCS*, pages 333–336. Springer, 2004.

[52] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The Diagnosis Frontend of the DLV System. *AI Communications*, 12(1-2):99–111, 1999.

[53] Omar Elkhatib, Enrico Pontelli, and Tran Cao Son. ASP-PROLOG: A System for Reasoning about Answer Set Programs in Prolog. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages (PADL 2004)*, volume 3057 of *LNCS*, pages 148–162. Springer, 2004.

[54] Enrico Ellguth, Martin Gebser, Markus Gusowski, Benjamin Kaufmann, Roland Kaminski, Stefan Liske, Torsten Schaub, Lars Schneidenbach, and Bettina Schnor. A Simple Distributed Conflict-Driven Answer Set Solver. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 490–495. Springer, 2009.

[55] Esra Erdem. PHYLO-ASP: Phylogenetic Systematics with Answer Set Programming. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 567–572. Springer, 2009.

[56] Esra Erdem, Ozan Erdem, and Ferhan Türe. HAPLO-ASP: Haplotype Inference Using Answer Set Programming. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 573–578. Springer, 2009.

[57] Esra Erdem and Vladimir Lifschitz. Tight Logic Programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.

[58] Esra Erdem, Vladimir Lifschitz, Luay Nakhleh, and Donald Ringe. Reconstructing the Evolutionary History of Indo-European Languages Using Answer

Set Programming. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, volume 2562 of *LNCS*, pages 160–176. Springer, 2003.

[59] European Commission Fifth Framework Programme (FP5). WASP: *Working Group on Answer Set Semantics* (IST-FET-2001-37004). `http://tinyurl.com/ist-wasp`, 2005. [accessed 2009-08-01].

[60] François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[61] Henry Falconer, Paul H.J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A Declarative Framework for Analysis and Optimization. In *Proceedings of the 16th International Conference on Compiler Construction (CC 2007)*, volume 4420 of *LNCS*, pages 218–232. Springer, 2007.

[62] Free Software Foundation. GCC, the GNU Compiler Collection. `http://gcc.gnu.org`. [accessed 2009-08-01].

[63] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. MILEPOST GCC: Machine Learning Based Research Compiler. In *Proceedings of the GCC Developers' Summit*, pages 7–19, 2008.

[64] Grigori Fursin and Olivier Temam. Collective Optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2009)*, volume 5409 of *LNCS*, pages 34–49. Springer, 2009.

[65] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[66] GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, 2009.

[67] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. *A Users Guide to* GRINGO, CLASP, CLINGO, *and* ICLINGO. University of Potsdam, 2008.

[68] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the Input Language of ASP Grounder GRINGO. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 502–508. Springer, 2009.

[69] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. CLASP: A Conflict-Driven Answer Set Solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.

[70] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczyński. The First Answer Set Programming System Competition. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *LNCS*, pages 3–17. Springer, 2007.

[71] Martin Gebser, Torsten Schaub, and Sven Thiele. GRINGO: A New Grounder for Answer Set Programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.

[72] Martin Gebser, Torsten Schaub, Sven Thiele, Bjrn Usadel, and Philippe Veber. Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 130–144. Springer, 2008.

[73] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[74] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[75] Paolo Giorgini, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Requirements Engineering for Trust Management: Model, Methodology and Reasoning. *International Journal of Information Security*, 5(4):257–274, 2006.

[76] Enrico Giunchiglia, Nicola Leone, and Marco Maratea. On the Relation Among Answer Set Solvers. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):169–204, 2008.

[77] David W. Goodwin and Kent D. Wilken. Optimal and Near-Optimal Global Register Allocations using 0-1 Integer Programming. *Software: Practice and Experience*, 26(8):929–965, 1996.

[78] Torbjörn Granlund. GNU Multiple Precision Arithmetic Library (GMP). `http://gmplib.org/`. [accessed 2009-08-01].

[79] 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, 1992.

[80] Torbjörn Granlund and Richard Kenner. GSO: the GNU Superoptimizer. `http://directory.fsf.org/project/superopt/`, 1995. [accessed 2009-08-01].

[81] Jean Gressmann, Tomi Janhunen, Robert E. Mercer, Torsten Schaub, Sven Thiele, and Richard Tichy. PLATYPUS: A Platform for Distributed Answer Set Solving. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *LNCS*, pages 227–239. Springer, 2005.

[82] Mary Hall, David Padua, and Keshav Pingali. Compiler Research: the Next 50 Years. *Communications of the ACM*, 52(2):60–67, 2009.

[83] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, 29(12):84–89, 1996.

[84] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[85] Carl Hewitt. PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot. A.I. Laboratory Memo AIM-168, Massachusetts Institute of Technology, 1970.

[86] Maarit Hietalahti, Fabio Massacci, and Ilkka Niemelä. DES: A Challenge Problem for Nonmonotonic Reasoning Systems. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR 2000)*, 2000.

[87] Kenneth Hoste and Lieven Eeckhout. COLE: Compiler Optimization Level Exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*, pages 165–174. ACM, 2008.

[88] IBM Corporation. *The Power PC Compiler Writer's Guide*. Warthman Associates, 1996.

[89] Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. An ASP-Based System for e-Tourism. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 368–381. Springer, 2009.

[90] Intel Corporation. Intel Compiler Suite. `http://www.intel.com/software/products/compilers`. [accessed 2009-08-01].

[91] Intel Corporation. *Intel Architecture Optimization Reference Manual*. Intel Corporation, 1999. 245127-001.

[92] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2009. 248966-018.

[93] Intel Corporation. *Intel C++ Compiler 11.1 User and Reference Guides*. Intel Corporation, 2009. 304968-023US.

[94] Tomi Janhunen. Removing Redundancy from Answer Set Programs. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 729–733. Springer, 2008.

[95] 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, 2002.

[96] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A Practical Algorithm for Generating Optimal Code. *ACM Transactions on Programming Languages and Systems*, 28(6):967–989, 2006.

[97] Samuel Kamin and Eric Golin. Report of a Workshop on Future Directions in Programming Languages and Compilers. *ACM SIGPLAN Notices*, 30(7):9–28, 1995.

[98] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Validation of GCC Optimizers Through Trace Generation. *Software: Practice and Experience*, 39(6):611–639, 2009.

[99] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1988.

[100] Christoph Keßler and Andrzej Bednarski. A Dynamic Programming Approach to Optimal Integrated Code Generation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 165–174. ACM, 2001.

[101] Peter B. Kessler. Discovering Machine-Specific Code Improvements. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN'86)*, pages 249–254. ACM, 1986.

[102] Robert R. Kessler. Peep – An Architectural Description Driven Peephole Optimizer. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN'84)*, pages 106–110. ACM, 1984.

[103] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105–133, 1970.

[104] Robert A. Kowalski. Predicate Logic as Programming Language. In *Proceedings of the International Federation of Information Processing Congress 74*, pages 569–574, 1974.

[105] Robert A. Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, 31(1):38–43, 1988.

[106] Robert A. Kowalski and Donald Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2:227–260, 1971.

[107] Ulrich Kremer. Optimal and Near-Optimal Solutions For Hard Compilation Problems. *Parallel Processing Letters*, 7(4):371–378, 1997.

[108] Viren Kumar and James Delgrande. Optimal Multicore Scheduling: An Application of ASP Techniques. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 604–609. Springer, 2009.

[109] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving Correctness of Compiler Optimizations by Temporal Logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 283–294. ACM, 2002.

[110] Scott Robert Ladd. ACOVEA: *Analysis of Compiler Options via Evolutionary Algorithm*. `http://www.coyotegulch.com/products/acovea/`. [accessed 2009-08-01].

[111] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[112] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proceedings of the First Annual GCC Developers' Summit*, pages 75–84, 2003.

[113] Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the Behavior of Compiler Optimizations. *Software: Practice and Experience*, 36(8):835–844, 2006.

[114] Claire Lefèvre and Pascal Nicolas. The First Version of a New ASP Solver: ASPERIX. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*. Springer, 2009.

[115] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, volume 2173 of *LNCS*, pages 280–294. Springer, 2001.

[116] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[117] Rainer Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms and Tools*. Springer, 2000.

[118] Chuck Liang. Compiler Construction in Higher Order Logic Programming. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, volume 2257 of *LNCS*, pages 47–63. Springer, 2002.

[119] Yuliya Lierler. Abstract Answer Set Solvers. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 377–391. Springer, 2008.

[120] Yuliya Lierler and Marco Maratea. CMODELS-2: SAT-based Answer Set Solver Enhanced to Non-Tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *LNCS*, pages 346–350. Springer, 2004.

[121] Vladimir Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[122] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[123] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A Compiler Framework for Speculative Analysis and Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03*, pages 289–299. ACM, 2003.

[124] John W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1993.

[125] LLVM Project. *clang*: a C Language Family Frontend for LLVM. `http://clang.llvm.org/`. [accessed 2009-08-01].

[126] D. W. Loveland. A Linear Format for Resolution. In *Proceedings of the Symposium on Automatic Declaration*, volume 125 of *Lecture Notes in Mathematics*, pages 147–162. Springer, 1970.

[127] Edward S. Lowry and C. W. Medlock. Object Code Optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[128] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek. An Application of Constraint Programming to Superblock Instruction Scheduling. In *Proceedings of th 14th International Conference of Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *LNCS*, pages 97–111. Springer, 2008.

[129] Toni Mancini, Davide Micaletto, Fabio Patrizi, and Marco Cadoli. Evaluating ASP and Commercial Solvers on the CSPLib. *Constraints*, 13(4):407–436, 2008.

[130] Victor Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[131] 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, 1987.

[132] John McCarthy. Programs with Common Sense. In *Semantic Information Processing*, pages 403–418. MIT Press, 1959.

[133] W. M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[134] Veena S. Mellarkod. Optimizing the Computation of Stable Models using Merged Rules. Master's thesis, Texas Tech University, Texas, USA, May 2002.

[135] Gordon Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

[136] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Conference on Design Automation (DAC'01)*, pages 530–535. ACM, 2001.

[137] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[138] Alan Mycroft. HAKMEMC – HAKMEM Programming Hacks in C. `http://www.cl.cam.ac.uk/users/am/hakmemc.html`. [accessed 2009-08-01].

[139] George Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 83–94. ACM, 2000.

[140] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. MIT Press, 1996.

[141] Ilkka Niemelä and Patrik 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 Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *LNCS*, pages 420–429. Springer, 1997.

[142] Ilkka Niemelä and Patrik Simons. *Logic-Based Artificial Intelligence*, chapter Extending the Smodels System with Cardinality and Weight Constraints, pages 491–521. Kluwer Academic Publishers, 2001.

[143] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An *A-Prolog* Decision Support System for the Space Shuttle. In *Proceedings of 3rd International Symposium on Practical Aspects of Declarative Languages (PADL 2001)*, volume 1990 of *LNCS*, pages 169–183. Springer, 2001.

[144] Open64 Project. Open64 - The Open Research Compiler. `http://www.open64.net/`. [accessed 2009-08-01].

[145] Mario Ornaghi, Camillo Fiorentini, Alberto Momigliano, and Francesco Pagano. Applying ASP to UML Model Validation. In *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 457–463. Springer, 2009.

[146] Zhelong Pan and Rudolf Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*, pages 319–332. IEE Computer Society, 2006.

[147] David A. Patterson and David R. Ditzel. The Case for the Reduced Instruction Set Computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[148] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2004.

[149] E. Pontelli, M. Balduccini, and F. Bermudez. Non-monotonic Reasoning on Beowulf Platforms. In *Proceedings of 5th International Symposium on Prac-*

tical Aspects of Declarative Languages (PADL 2003), volume 2562 of *LNCS*, pages 37–57. Springer, 2003.

[150] Enrico Pontelli, Hung Viet Le, and Tran Cao Son. An Investigation in Parallel Execution of Answer Set Programs on Distributed Memory Platforms: Task Sharing and Dynamic Scheduling. *Computer Languages, Systems & Structures*, 36(2):158–202, 2010.

[151] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, 2008.

[152] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156. IEE Computer Society, 2007.

[153] President's Information Technology Advisory Committee (PITAC). Computational Science: Ensuring America's Competitiveness. Technical report, Executive Office of the President of the United States, June 2005.

[154] Todd A. Proebsting. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. `http://research.microsoft.com/en-us/um/people/toddpro/papers/law.htm`. [accessed 2009-08-01].

[155] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[156] Arch D. Robinson. Impact of Economics on Compiler Optimization. In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande (JGI'01)*, pages 1–10. ACM, 2001.

[157] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[158] Roger Anthony Sayle. A Superoptimizer Analysis of Multiway Branch Code Generation. In *Proceedings of the GCC Summit*, pages 103–116, 2008.

[159] Paul B. Schneck. A Survey of Compiler Optimization Techniques. In *Proceedings of the ACM Annual Conference*, pages 106–113, 1973.

[160] Kevin Scott. On Proebsting's Law. Technical Report CS-2001-12, Department of Computer Science, University of Virginia, 2001.

[161] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[162] Daniel Serpell. SuperOptimizer for Microchip's PIC Microcontrollers. `http://tinyurl.com/pic-superoptimizer`. [accessed 2009-08-01].

[163] Ravi Sethi and Jeffrey D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[164] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland, April 2000.

[165] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[166] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*, 1992. Revision SAV080SI9308.

[167] Diomidis Spinellis. Declarative Peephole Optimization using String Pattern Matching. *ACM SIGPLAN Notices*, 34(2):47–50, 1999.

[168] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 77–90. ACM, 2003.

[169] Sun Developer Network. The Java HotSpot Performance Engine Architecture. Technical report, Sun Microsystems, 2008.

[170] Sun Microsystems, Inc. *SPARC Assembly Language Reference Manual*, 2002. Version 816168110.

[171] Tommi Syrjänen. *Lparse 1.0 User's Manual*. Helsinki University of Technology, 2007.

[172] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, 1982.

[173] Ross Tate, Michael Stepp, and Sorin Lerner. Generating Compiler Optimizations from Proofs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 389–402, 2010.

[174] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: a New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 264–276. ACM, 2009.

[175] The Stanford SUIF Compiler Group. SUIF Compiler System. `http://suif.stanford.edu/`. [accessed 2009-08-01].

[176] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral Code Generation in the Real World. In *Proceedings of the 15th International Conference on Compiler Construction (CC 2006)*, volume 3923 of *LNCS*, pages 185–201. Springer, 2006.

[177] Jürgen Vollmer. Experiences with *Gentle*: Efficient Compiler Construction based on Logic Programming. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, volume 528 of *LNCS*. Springer, 1991.

[178] Jeffrey Ward and John S. Schlipf. Answer Set Programming with Clause Learning. In *Proceedings of 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *LNCS*, pages 302–313. Springer, 2004.

[179] David H. D. Warren. Logic Programming and Compiler Writing. *Software: Practice and Experience*, 10(2):97–125, 1980.

[180] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc, 1994. Revision SAV09R1459912.

[181] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144. ACM, 2004.

[182] Kent Wilken, Jack Liu, and Mark Hefferman. Optimal Instruction Scheduling using Integer Programming. In *Proceedings of the ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI'00), pages 121–133. ACM, 2000.

[183] Gareth Williams. *Linear Algebra with Applications*. Jones and Bartlett, 4th edition, 2001.

[184] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. Predicting the Impact of Optimizations for Embedded Systems. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, pages 1–11. ACM, 2003.

Appendices

Appendix A

AnsProlog Language Description

A.1 Introduction

In the following section, we present an informal description of the *AnsProlog* language. No formal grammar or Extended BNF for the language has yet been defined, but there are a number of widely-accepted standards to which the domain tools adhere. The language that the LPARSE/SMODELS family of ASP tools accept [141, 171] is the most commonly implemented (referred to as *AnsProlog*^{sm}); this is accepted (and extended) by the GRINGO/CLASP family of ASP tools [67, 68, 71]. The DLV system [52, 116] accepts a different subset of the *AnsProlog* language (which includes disjunction in the head) and is not currently compatible with the other domain tools.

A.2 Syntax Conventions

The generally-accepted syntax conventions for *AnsProlog* are as follows:

Predicate symbols are composed of lower-case letters and digits and must start with a letter.

Constant symbols are either just digits or composed of lower-case letters and digits, starting with a lower-case letter.

Variable symbols are composed of letters and digits and must start with an upper-case letter.

A fact is followed by a dot '.'.

Rules use the following symbols and are followed by a dot '.'.

- Negation as failure: '`not`'
- If ($\leftarrow$): '`:-`'
- Conjunction ($\wedge$): '`,`'

- Disjunction ($\vee$): '`;`'
- Classical negation ($\neg$): '`-`'

Arithmetic comparisons use the following symbols:

- Equality: '`==`'
- Inequality: '`!=`'
- Less than or equal: '`<=`'
- Greater than or equal: '`>=`'
- Less than: '`<`'
- Greater than: '`>`'

Appendix B

AnsProlog Literals in the TOAST System

B.1 Introduction

This is a definitive list of all *AnsProlog* literals used within the TOAST system, describing their format, arguments and usage.

B.2 Literals

- `arg(R)`
  `R` is a register or it is `none`.
- `asymmetricInstruction(I)`
  Instruction `I` takes two arguments, order irrelevant.
- `before(R1,R2)`
  An aribtrary ordering on the register (input and output) so that re-orderings can be removed.
- `bit(B)`
  Each bit of a register is modelled independently.
- `bitOrExtended(B)`
  A copy of `bit(B)` with additional values for -1 $\rightarrow$ -`wordLength`, basically zero padding the lower part of a register so shifted subtracts (in divide) work correctly.
- `colour(C)`
  Each instruction stream is labeled by a colour, the following are in use: `red` - `execute` and `verify`; the initial code sequence, `blue` - `search` and `verify`; the new code sequence, `purple` - `verify`; used to save on rule generation.

- `constrained(C,CT,R)`
  True if register `R` in stream `C` is used constraint `CT`.

- `dividend(C,T,B)`
  Register for the sign removed version of the first argument of a signed multiply.

- `dividendCarry(C,T,B)`
  Similar to above, but with carry.

- `divideSubCycle(C,T,S)`
  True if subcycle `S` is needed to compute a division.

- `divideSubCycleCount(C,T,B,S)`
  A counter used to correctly work out what `divideSubCycle(C,T,S)` should be set to.

- `divisor(C,T,B)`
  Second argument of signed multiply.

- `divisorCarry(C,T,B)`
  Similar to above, but with carry.

- `enableConstraints`
  Trivial atom used to make generating rules easier in `buildMultiple`.

- `extended(E)`
  A workaround to get `lparse` to generate the correct values for `bitOrExtended`.

- `finishedAt(C,T)`
  The time, `T`, at which stream `C` executes its last valid instruction (inclusive). Currently used in `execute` and `verify`.

- `flag(F)`
  `F` is a processor flag.

- `flagChanged(C,T,F)`
  True is flag `F` has been altered by the instruction run at time step `T` in stream `C`.

- `flagStatus(C,T,F)`
  True if flag `F` is set at time `T` in stream `C`, true negated if it is set to 0.

- `flagUseful(C,F,P1,P2)`
  True if flag `F` (set at `P1`) is still usable at `P2` (in stream `C`). Used for in optimisations in `search`.

- `haveJumped(C,T)`
  Whether a jump has occured at in stream `C` at time `T` (product of jump instruction, defined by flow control).

- `hiOutput(I)`
  True if instruction `I` gives a high output (i.e. [`wordLength`, 2*`wordLength`-1]), accessible with `hi`.

- `immediate(R)`
  Register `R` is an immediate.

- `input(R)`
  Register `R` is an input.

- `instruction(I)`
  `I` is an instruction.

- `instructionChoice(C,T,I)`
  At time `T` in stream `C`, there is an instruction `I`.

- `instructionUsed(C,P)`
  True if the instruction at position `P` in stream `C` is used. Used for optimisations in `search`.

- `istream(C,P,IN,R1,R2,R3)`
  Instruction stream `C`, at position `P` is instruction `IN` using registers `R1` and `R2` (may also have the value `none`). `R3` is an optional third argument, used in execution only.

- `jump(C,T,J)`
  When executing stream `C` at time `T`, jump `J` positions forward, asserted by instructions.

- `jumpSize(C,J)`
  The possible sizes of jump, 2 to (`lengthOfInstructionSequence` - 1)

- `leadingZero(C,T,B)`
  A means for discounting the leading zeros when counting the number of sub cycles needed to divide.

- `lessThanInDivide(C,T,B,S)`
  True with `B` = 0 if the divisor is less than or equal to the `subCycle` value; for `B` != 0, used to calculate.

- `mayJump(C,T,J)`
  True if it is possible there is a jump of `J` at time `T` in stream `C`.

- `negateResult(C,T)`
  Where the output of the multiply executed in `C` at time `T` needs to be negated or not.

- `negateResultCarry(C,T)`
  Used to negate the result of a signed multiply.

- `noArgInstruction(I)`
  Instruction `I` takes no arguments

- `nonEquivalent`
  The code streams red and blue are not equivalent.

- `output(R)`
  Register `R` is an output.

- `outputConstraints(CT)`
  The number of outputs on which two code streams have to match (in `verify`), numbered from 1 in the order they appear in the program.

- `pc(C,PCV,T)`
  The program counter for stream `C` has value `PCV` at time `T`.

- `pcValue(C,PCV)`
  The possible values of the program counter, position + `jumpSize` (set addition), so 1 to (2 * `lengthOfInstructionSequence` - 1).

- `position(C,P)`
  Indexes `istream C`, counting from 1 to `lengthOfInstructionSequence`.

- `predicateVar(C,T,B)`
  The second argument of a signed multiplication after sign removal.

- `range(E)`
  A workaround to get `lparse` to generate the corretc values for `extended` and `bitOrExtended`.

- `register(R)`
  These are the registers, `iX` are input vectors, `X` are created during execution. `i0` is the hardwired register (if available), `X` starts at 1.

- `registerDefined(C,R)`
  True if register `R` is defined in stream colour `C`.

- `relevantFlag(F)`
  True if the value of the given flag at the end of the program is deemed to be important.

- `requireValidSequence`
  Trivial atom used to make generation in `searchCut` easier

- `runTimeError(C,T)`
  The instruction executed at time `T` in stream `C` causes a run time error (i.e. it is poorly defined). These are mostly synthesised now.

- `setsFlag(I,F)`
  True if instruction `I` may set flag `F`.

- `subCycle(C,T,B,S)`
  A variant on value, true if bit `B` of sub cycle `S` at time `T` (in stream `C`) is 1, true negated if it is 0.

- `subCycleBorrow(C,T,B,S)`
  Used to denote the carry bits in the subtraction within divide.

- `subtractionBorrow(C,T,B)`
  The borrow register of the subtraction implementation.

- `sumVar(C,T,B)`
  The first argument of a signed multiplication after sign removal.

- `sumVarCarry(C,T,B)`
  Used to convert to `sumVar`.

- `symmetricInstruction(I)`
  Instruction `I` takes two arguments, order is irrelevant.

- `time(C,T)`
  The steps of a program's execution, bounded by program size. Stream dependant.

- `unaryInstruction(I)`
  Instruction `I` takes one argument.

- `usesFlag(I,F)`
  True if instruction `I` may use flag `F`.

- `validSequence`
  Trivial atom used by `searchCut` to reduce the search space covered by `search`.

- `value(C,R,B)`
  The value of bit `B` of register `R` in instruction stream `C`, positive for 1, true negated for 0.

Appendix C

TOAST Architecture Descriptions

C.1 Introduction

In the following sections, we list the full TOAST architecture descriptions for the following architectures: MIPS R2000, SPARC V7 and SPARC V8.

Each architecture has its own file that defines general information, the available instructions and to which part of the TOAST system they are applicable. It has one declaration per line, with columns separated by two tabs. A declaration is either an option (two column) or a mapping (three column). Comments are denoted by a # character and are to the end of the line.

The initial options section defines the number of architecture bits, its family and name e.g. `risc` and `sparc-v8`. zero is whether the architecture has a hardwired zero register. Flags define which condition codes are present and how they are handled.

The instruction section takes the following form:

```
[assembly]    {search,exec,combo,undef}    [internal form]
```

The `[assembly]` column is the machine-specific assembly language name. This can be '-', which means that the internal form is included in the search space. However, the caveat of this encoding is that if it is picked in any combination it will not map it back unless it also appears in a combo declaration (explained below).

`search` means that the internal form (single instruction) should be included in both the search and the execute space.

`exec` means that the internal form (single instruction) should be included in the execute space.

`combo` mean that the internal form is multiple instructions. There must also exist individual search or exec definitions for this to be valid.

`undef` mean that the internal form is undefined. There should be an explanation of why the instruction is not defined.

The `[internal form]` column is a pattern of one or more internal TOAST opcodes (separated by semi-colons) with directions for how the arguments and instruction outputs are modelled. For example, `a1, a2, a3` should be substituted for the first, second and third arguments of the assembly instruction (once converted to the appropriate internal TOAST form); `n` is the location of the first instruction in the sequence, `n+1` the second, and so on. `o1, o2, o3` are the outputs of the first, second and third opcodes.

C.2 MIPS R2000

Architectural and instruction information taken from the *MIPS RISC Architecture* book [99]; a brief overview of the MIPS R2000 architecture can also be found in Section 5.3 (page 44).

```
## MIPS R2000 processor
## General info
bits        32
zero        yes
family      risc
name        mips-r2000
flags       none

## Instructions
# Assembler is Intel-style : destination, source, source
add         undef       May trigger overflow exception
addi        undef       May trigger overflow exception
addiu       search      add a2 a3
addu        search      add a2 a3
and         search      land a2 a3
andi        search      land a2 a3
bczf        undef       Co-processor is implementation defined
bczt        undef       Co-processor is implementation defined
beq         exec        beq a1 a2 a3
bgez        exec        bgez a1 a2
bgezal      undef       Puts address in register
bgtz        exec        bgtz a1 a2
blez        exec        blez a1 a2
bltz        exec        bltz a1 a2
bltzal      undef       Puts address in register
bne         exec        bne a1 a2 a3
break       undef       Triggers execption
cfcz        undef       Co-processor is implementation defined
copz        undef       Co-processor is implementation defined
ctcz        undef       Co-processor is implementation defined
div         search      sdiv a1 a2
divu        search      udiv a1 a2
j           exec        br a1
jal         undef       Puts address in register
jalr        undef       Puts address in register
```

```
jr          undef       Mixes register values and addresses
lb          undef       Load
lbu         undef       Load
lh          undef       Load
lhu         undef       Load
lui         undef       Load
lw          undef       Load
lwz         undef       Co-processor is implementation defined
lwl         undef       Patented
lwr         undef       Patented
mfcz        undef       Co-processor is implementation defined
mfhi        search      hi a1
mflo        undef       Translation implemented
mtcz        undef       Co-processor is implementation defined
mthi        undef       Translation implemented
mtlo        undef       Translation implemented
mult        search      smult a1 a2
multu       search      umult a1 a2
nor         search      lnor a2 a3
or          search      lor a2 a3
ori         search      lor a2 a3
rfe         undef       Non local jump
sb          undef       Store
sh          undef       Store
sll         search      slz32 a2 a3
sllv        search      slz32 a2 a3
slt         search      slt a2 a3
slti        search      slt a2 a3
sltiu       search      sltu a2 a3
sltu        search      sltu a2 a3
sra         search      srs32 a2 a3
srav        search      srs32 a2 a3
srl         search      srz32 a2 a3
srlv        search      srz32 a2 a3
sub         undef       May trigger overflow exception
subu        search      sub a2 a3
sw          undef       Store
swcz        undef       Store
swl         undef       Patented
swr         undef       Patented
syscall     undef       Non local jump
tlbp        undef       TLB operation
tlbr        undef       TLB operation
tlbwi       undef       TLB operation
tlbwr       undef       TLB operation
xor         search      lxor a2 a3
xori        search      lxor a2 a3

# Generic definitions needed by an number of instructions
-           exec        equal
-           exec        greaterThanZero
```

```
87  -           exec        isZero
```

Listing C.1: MIPS R2000 architecture description

C.3 SPARC V7

Architectural and instruction information taken from the *SPARC V7 Instruction Set* manual [12]; a brief overview of the SPARC V7 architecture can also be found in Section 6.2 (page 64).

```
1   ## General info
2   bits        32
3   zero        yes
4   family      risc
5   name        sparc-v7
6   flags       neg zero over carry
7
8   ## Instructions
9   # Assembler is AT&T style : source, source, destination
10
11  # B.1 Load Integer Instructions
12  ldsb        undef       Load
13  ldsba       undef       Load, privileged
14  ldsh        undef       Load
15  ldsha       undef       Load, privileged
16  ldub        undef       Load
17  lduba       undef       Load, privileged
18  lduh        undef       Load
19  lduha       undef       Load, privileged
20  ld          undef       Load
21  lda         undef       Load, privileged
22  ldd         undef       Load
23  ldda        undef       Load, privileged
24
25  # B.2 Load Floating-point Instructions
26  ldf         undef       Load, floating point
27  lddf        undef       Load, floating point
28  ldfsr       undef       Load, floating point
29
30  # B.3 Load Coprocessor Instructions
31  ldc         undef       Load, coprocessor, implementation
        dependent
32  lddc        undef       Load, coprocessor, implementation
        dependent
33  ldcsr       undef       Load, coprocessor, implementation
        dependent
34
35  # B.4 Store Integer Instructions
```

```
stb         undef       Store
stba        undef       Store, privileged
sth         undef       Store
stha        undef       Store, privileged
st          undef       Store
sta         undef       Store, privileged
std         undef       Store
stda        undef       Store, privileged

# B.5 Store Floating-point Instructions
stf         undef       Store
stdf        undef       Store
stfsr       undef       Store
stdfq       undef       Store, privileged

# B.6 Store Coprocessor Instructions
stc         undef       Store, coprocessor, implementation
   dependent
stdc        undef       Store, coprocessor, implementation
   dependent
stcsr       undef       Store, coprocessor, implementation
   dependent
stdcq       undef       Store, coprocessor, implementation
   dependent, privil.

# B.7 Atomic Load-Store Unsigned Byte Instructions
ldstub      undef       Load, store
ldstuba     undef       Load, store, privileged

# B.8 SWAP r Register with Memory Instruction
swap        undef       Load, store
swapa       undef       Load, store, privileged

# B.9 Add Instructions
# one argument can be a 13 bit immediate, which is then sign
   extended
add         search      add a1 a2
addcc       search      addcc a1 a1
addx        search      cadd a1 a2
addxcc      search      caddcc a1 a2

# B.10 Tagged Add Instructions
# one argument can be a 13 bit immediate, which is then sign
   extended
taddcc      search      taddcc a1 a1
taddcctv    undef       May trap

# B.11 Subtract Instructions
# one argument can be a 13 bit immediate, which is then sign
   extended
sub         search      sub a1 a2
subcc       search      subcc a1 a2
subx        search      csub a1 a2
```

```
82  subxcc      search      csubcc a1 a2
83
84  # B.12 Tagged Subtract Instructions
85  # one argument can be a 13 bit immediate, which is then sign
       extended
86  tsubcc      search      tsubcc a1 a2
87  tsubcctv    undef       May trap
88
89  # B.13 Multiply Step Instruction
90  # one argument can be a 13 bit immediate, which is then sign
       extended
91  #mulscc     search      multstcc a1 a2
92  mulscc      undef       Temporary fix
93
94  # B.14 Logical Instructions
95  # one argument can be a 13 bit immediate, which is then sign
       extended
96  and         search      land a1 a2
97  andcc       search      landcc a1 a2
98  andn        combo       lnot a2; land a1 o1
99  andncc      combo       lnot a2; landcc a1 o1
100 or          search      lor a1 a2
101 orcc        search      lorcc a1 a2
102 orn         combo       lnot a2; lor a1 o1
103 orncc       combo       lnot a2; lorcc a1 o1
104 xor         search      lxor a1 a2
105 xorcc       search      lxorcc a1 a2
106 xnor        combo       lxor a1 a2 ; lnot o1
107 xnorcc      combo       lnot a2; lxorcc a1 o1
108 -           search      lnot
109 -           search      isZero
110
111 # B.15 Shift Instructions
112 # one argument can be a 13 bit immediate, which is then sign
       extended
113 sll         search      slz32
114 srl         search      srz32
115 sra         search      srs32
116
117 # B.16 SETHI Instruction
118 # this is slightly problematic as it can / only/ take an
       immediate as an argument
119 sethi       exec        sethi a1
120
121 # B.17 SAVE and RESTORE Instructions
122 save        undef       Requires modelling of specific registers
123 restore     undef       Requires modelling of specific registers
124
125 # B.18 Branch on Integer Condition Codes Instructions
126 ba          exec        cba a1
127 bn          exec        cbn a1
128 bne         exec        cbne a1
129 be          exec        cbe a1
```

```
bg          exec        cbg a1
ble         exec        cble a1
bge         exec        cbge a1
bl          exec        cbl a1
bgu         exec        cbgu a1
bleu        exec        cbleu a1
bcc         exec        cbcc a1
bcs         exec        cbcs a1
bpos        exec        cbpos a1
bneg        exec        cbneg a1
bvc         exec        cbvc a1
bvs         exec        cbvs a1

# B.19 Branch on Floating-point Condition Codes Instructions
fba         undef       Floating point
fbn         undef       Floating point
fbu         undef       Floating point
fbg         undef       Floating point
fbug        undef       Floating point
fbl         undef       Floating point
fbul        undef       Floating point
fblg        undef       Floating point
fbne        undef       Floating point
fbe         undef       Floating point
fbue        undef       Floating point
fbge        undef       Floating point
fbuge       undef       Floating point
fble        undef       Floating point
fbule       undef       Floating point
fbo         undef       Floating point

# B.20 Branch on Coprocessor Condition Codes Instructions
cba         undef       Coprocessor, implementation dependent
cbn         undef       Coprocessor, implementation dependent
cb3         undef       Coprocessor, implementation dependent
cb2         undef       Coprocessor, implementation dependent
cb23        undef       Coprocessor, implementation dependent
cb1         undef       Coprocessor, implementation dependent
cb13        undef       Coprocessor, implementation dependent
cb12        undef       Coprocessor, implementation dependent
cb123       undef       Coprocessor, implementation dependent
cb0         undef       Coprocessor, implementation dependent
cb03        undef       Coprocessor, implementation dependent
cb02        undef       Coprocessor, implementation dependent
cb023       undef       Coprocessor, implementation dependent
cb01        undef       Coprocessor, implementation dependent
cb013       undef       Coprocessor, implementation dependent
cb012       undef       Coprocessor, implementation dependent

# B.21 Call and Link Instruction
call        undef       Position dependent

# B.22 Jump and Link Instruction
```

```
jmpl        undef       Position dependent

# B.23 Return from Trap Instruction
rett        undef       privileged

# B.24 Trap on Integer Condition Codes Instruction
ta          undef       Trap
tn          undef       Trap
tne         undef       Trap
te          undef       Trap
tg          undef       Trap
tle         undef       Trap
tge         undef       Trap
tl          undef       Trap
tgu         undef       Trap
tleu        undef       Trap
tcc         undef       Trap
tcs         undef       Trap
tpos        undef       Trap
tneg        undef       Trap
tvc         undef       Trap
tvs         undef       Trap

# B.25 Read State Register Instructions
rdy         search      hi
rdasr       undef       privileged, implementation dependent
rdpsr       undef       privileged
rdwim       undef       privileged
rdtbr       undef       privileged

# B.26 Write State Register Instructions
wry         undef       You /shouldn't/ need to do this
wrasr       undef       privileged, implementation dependent
wrpsr       undef       privileged
wrwim       undef       privileged
wrtbr       undef       privileged

# B.27 Unimplemented Instruction
unimp       undef       And neither will we...

# B.28 Flush Instruction Memory
iflush      undef       Self modifying code!

# B.29 Floating-point Operate (FPop) Instructions
fpop1       undef       Floating point
fpop2       undef       Floating point
fitos       undef       Floating point
fitod       undef       Floating point
fitox       undef       Floating point
fstoi       undef       Floating point
fdtoi       undef       Floating point
fxtoi       undef       Floating point
fstod       undef       Floating point
```

```
fstox       undef       Floating point
fdtos       undef       Floating point
fdtox       undef       Floating point
fxtos       undef       Floating point
fxtod       undef       Floating point
fmov        undef       Floating point
fnegs       undef       Floating point
fabs        undef       Floating point
fsqrts      undef       Floating point
fsqrtd      undef       Floating point
fsqrtx      undef       Floating point
fadds       undef       Floating point
faddd       undef       Floating point
faddx       undef       Floating point
fsubs       undef       Floating point
fsubd       undef       Floating point
fsubx       undef       Floating point
fmuls       undef       Floating point
fmuld       undef       Floating point
fmulx       undef       Floating point
fsmuld      undef       Floating point
fdmulx      undef       Floating point
fdivs       undef       Floating point
fdivd       undef       Floating point
fdivx       undef       Floating point
fcmps       undef       Floating point
fcmpd       undef       Floating point
fcmpx       undef       Floating point
fcmpes      undef       Floating point
fcmped      undef       Floating point
fcmpex      undef       Floating point
fcmpex      undef       Floating point

# B.34 Coprocessor Operate Instructions
cpop1       undef       Coprocessor, implementation dependent
cpop2       undef       Coprocessor, implementation dependent
```

Listing C.2: SPARC V7 architecture description

C.4 SPARC V8

Architectural and instruction information taken from the *SPARC V8 Instruction Set* manual [166]; a brief overview of the SPARC V8 architecture can also be found in Section 6.2 (page 64).

```
## SPARC V8 processor
# General info
bits        32
```

```
zero        yes
family      risc
name        sparc-v8
flags       neg zero over carry

## Instructions
# Assembler is AT&T style: source, source, destination

# B.1 Load Integer Instructions
ldsb        undef       Load
ldsh        undef       Load
ldub        undef       Load
lduh        undef       Load
ld          undef       Load
ldd         undef       Load
ldsba       undef       Load, privileged
ldsha       undef       Load, privileged
lduba       undef       Load, privileged
lduha       undef       Load, privileged
lda         undef       Load, privileged
ldda        undef       Load, privileged

# B.2 Load Floating-point Instructions
ldf         undef       Load, floating point
lddf        undef       Load, floating point
ldfsr       undef       Load, floating point

# B.3 Load Coprocessor Instructions
ldc         undef       Load, coprocessor, implementation
   dependent
lddc        undef       Load, coprocessor, implementation
   dependent
ldcsr       undef       Load, coprocessor, implementation
   dependent

# B.4 Store Integer Instructions
stb         undef       Store
sth         undef       Store
st          undef       Store
std         undef       Store
stba        undef       Store, privileged
stha        undef       Store, privileged
sta         undef       Store, privileged
stda        undef       Store, privileged

# B.5 Store Floating-point Instructions
stf         undef       Store
stdf        undef       Store
stfsr       undef       Store
stdfq       undef       Store, privileged

# B.6 Store Coprocessor Instructions
```

```
53 stc         undef       Store, coprocessor, implementation
      dependent
54 stdc        undef       Store, coprocessor, implementation
      dependent
55 stcsr       undef       Store, coprocessor, implementation
      dependent
56 stdcq       undef       Store, coprocessor, implementation
      dependent, privil.
57
58 # B.7 Atomic Load-Store Unsigned Byte Instructions
59 ldstub      undef       Load, store
60 ldstuba     undef       Load, store, privileged
61
62 # B.8 SWAP Register with Memory Instruction
63 swap        undef       Load, store
64 swapa       undef       Load, store, privileged
65
66 # B.9 SETHI Instruction
67 # can only take an immediate as an argument
68 sethi       exec        sethi a1
69
70 # B.10 NOP Instruction
71 nop         exec        nop
72
73 # B.11 Logical Instructions
74 # one argument can be a 13 bit immediate, which is then sign
      extended
75 and         search      land a1 a2
76 andcc       search      landcc a1 a2
77 andn        combo       lnot a2; land a1 o1
78 andncc      combo       lnot a2; landcc a1 o1
79 or          search      lor a1 a2
80 orcc        search      lorcc a1 a2
81 orn         combo       lnot a2; lor a1 o1
82 orncc       combo       lnot a2; lorcc a1 o1
83 xor         search      lxor a1 a2
84 xorcc       search      lxorcc a1 a2
85 xnor        combo       lxor a1 a2 ; lnot o1
86 xnorcc      combo       lnot a2; lxorcc a1 o1
87 -           search      lnot
88 -           search      isZero
89
90 # B.12 Shift Instructions
91 # one argument can be a 13 bit immediate, which is then sign
      extended
92 sll         search      slz32
93 srl         search      srz32
94 sra         search      srs32
95
96 # B.13 Add Instructions
97 # one argument can be a 13 bit immediate, which is then sign
      extended
98 add         search      add a1 a2
```

```
addcc       search      addcc a1 a1
addx        search      cadd a1 a2
addxcc      search      caddcc a1 a2

# B.14 Tagged Add Instructions
# one argument can be a 13 bit immediate, which is then sign
    extended
taddcc      search      taddcc a1 a1
taddcctv    undef       May trap

# B.15 Subtract Instructions
# one argument can be a 13 bit immediate, which is then sign
    extended
sub         search      sub a1 a2
subcc       search      subcc a1 a2
subx        search      csub a1 a2
subxcc      search      csubcc a1 a2

# B.16 Tagged Subtract Instructions
# one argument can be a 13 bit immediate, which is then sign
    extended
tsubcc      search      tsubcc a1 a2
tsubcctv    undef       May trap

# B.17 Multiply Step Instruction
# one argument can be a 13 bit immediate, which is then sign
    extended
#mulscc     search      multstcc a1 a2
mulscc      undef       Temporary fix

# B.18 Multiply Instructions
# one argument can be a 13 bit immediate, which is then sign
    extended
umul        search      umult a1 a2
smul        search      smult a1 a2
umulcc      search      umultcc a1 a2
smulcc      search      smultcc a1 a2


# B.19 Divide Instructions
# one argument can be a 13 bit immediate, which is then sign
    extended
udiv        undef       May trap
sdiv        undef       May trap
udivcc      undef       May trap
sdivcc      undef       May trap

# B.20 SAVE and RESTORE Instructions
save        undef       Requires modelling of specific registers
restore     undef       Requires modelling of specific registers

# B.21 Branch on Integer Condition Codes Instructions
ba          exec        cba a1
```

```
bn          exec        cbn a1
bne         exec        cbne a1
be          exec        cbe a1
bg          exec        cbg a1
ble         exec        cble a1
bge         exec        cbge a1
bl          exec        cbl a1
bgu         exec        cbgu a1
bleu        exec        cbleu a1
bcc         exec        cbcc a1
bcs         exec        cbcs a1
bpos        exec        cbpos a1
bneg        exec        cbneg a1
bvc         exec        cbvc a1
bvs         exec        cbvs a1

# B.22 Branch on Floating-point Condition Codes Instructions
fba         undef       Floating point
fbn         undef       Floating point
fbu         undef       Floating point
fbg         undef       Floating point
fbug        undef       Floating point
fbl         undef       Floating point
fbul        undef       Floating point
fblg        undef       Floating point
fbne        undef       Floating point
fbe         undef       Floating point
fbue        undef       Floating point
fbge        undef       Floating point
fbuge       undef       Floating point
fble        undef       Floating point
fbule       undef       Floating point
fbo         undef       Floating point

# B.23 Branch on Coprocessor Condition Codes Instructions
cba         undef       Coprocessor, implementation dependent
cbn         undef       Coprocessor, implementation dependent
cb3         undef       Coprocessor, implementation dependent
cb2         undef       Coprocessor, implementation dependent
cb23        undef       Coprocessor, implementation dependent
cb1         undef       Coprocessor, implementation dependent
cb13        undef       Coprocessor, implementation dependent
cb12        undef       Coprocessor, implementation dependent
cb123       undef       Coprocessor, implementation dependent
cb0         undef       Coprocessor, implementation dependent
cb03        undef       Coprocessor, implementation dependent
cb02        undef       Coprocessor, implementation dependent
cb023       undef       Coprocessor, implementation dependent
cb01        undef       Coprocessor, implementation dependent
cb013       undef       Coprocessor, implementation dependent
cb012       undef       Coprocessor, implementation dependent

# B.24 Call and Link Instruction
```

```
call        undef       Position dependent

# B.25 Jump and Link Instruction
jmpl        undef       Position dependent

# B.26 Return from Trap Instruction
rett        undef       privileged

# B.27 Trap on Integer Condition Codes Instruction
ta          undef       Trap
tn          undef       Trap
tne         undef       Trap
te          undef       Trap
tg          undef       Trap
tle         undef       Trap
tge         undef       Trap
tl          undef       Trap
tgu         undef       Trap
tleu        undef       Trap
tcc         undef       Trap
tcs         undef       Trap
tpos        undef       Trap
tneg        undef       Trap
tvc         undef       Trap
tvs         undef       Trap

# B.28 Read State Register Instructions
rdy         search      hi
rdasr       undef       privileged, implementation dependent
rdpsr       undef       privileged
rdwim       undef       privileged
rdtbr       undef       privileged

# B.29 Write State Register Instructions
wry         undef       You should not need to do this
wrasr       undef       privileged, implementation dependent
wrpsr       undef       privileged
wrwim       undef       privileged
wrtbr       undef       privileged

# B.30 STBAR Instruction
stbar       undef       Memory subsystem only

# B.31 Unimplemented Instruction
unimp       undef       And neither will we...

# B.32 Flush Instruction Memory
flush       undef       Self modifying code!

# B.33 Floating-point Operate (FPop) Instructions
fpop1       undef       Floating point
fpop2       undef       Floating point
fitos       undef       Floating point
```

```
fitod       undef       Floating point
fitoq       undef       Floating point
fstoi       undef       Floating point
fdtoi       undef       Floating point
fqtoi       undef       Floating point
fstod       undef       Floating point
fstoq       undef       Floating point
fdtos       undef       Floating point
fdtoq       undef       Floating point
fqtos       undef       Floating point
fqtod       undef       Floating point
fmov        undef       Floating point
fnegs       undef       Floating point
fabs        undef       Floating point
fsqrts      undef       Floating point
fsqrtd      undef       Floating point
fsqrtq      undef       Floating point
fadds       undef       Floating point
faddd       undef       Floating point
faddq       undef       Floating point
fsubs       undef       Floating point
fsubd       undef       Floating point
fsubq       undef       Floating point
fmuls       undef       Floating point
fmuld       undef       Floating point
fmulq       undef       Floating point
fsmuld      undef       Floating point
fdmulq      undef       Floating point
fdivs       undef       Floating point
fdivd       undef       Floating point
fdivq       undef       Floating point
fcmps       undef       Floating point
fcmpd       undef       Floating point
fcmpq       undef       Floating point
fcmpes      undef       Floating point
fcmped      undef       Floating point
fcmpeq      undef       Floating point
fcmpeq      undef       Floating point

# B.34 Coprocessor Operate Instructions
cpop1       undef       Coprocessor, implementation dependent
cpop2       undef       Coprocessor, implementation dependent
```

Listing C.3: SPARC V8 architecture description