Palamos House #208
66-67 High Street
Lymington
SO41 9AL United Kingdom
www.embecosm.com

Superoptimization

Feasibility Study by Embecosm Limited, supported by Innovate UK

Superoptimization is the process of finding the optimal instruction sequence for a given section of code. This contrast with the traditional process of compilation, which optimizes code but does not ensure optimality.

This report explores the different types of superoptimizer, and how they could be implemented in modern technologies.

This work was led by James Pallister, Research Engineer at Embecosm and supported by Innovate UK under their Technology Inspired Feasibility Studies initiative.

For further information about this work, please contact Dr Jeremy Bennett, Chief Executive, jeremy.bennett@embecosm.com.

1 Introduction

Superoptimization is an idea which produces perfectly optimal code, in place of the code we currently have generated by compilers. This is typically done via a brute-force search of every possible instruction sequence, checking whether it performs the desired actions and accepting the sequence if it is the optimal one.

The problem is clearly very difficult, exploding in size as the length of the instruction sequence increases. However, superoptimization gaining traction as a method of optimizing programs. In this feasibility study we hope to find which techniques are extensible enough to bring out of the academic community to be using in a commercial setting. This means that the superoptimizer needs to be able to handle all the corner cases that may arise in production code. If it turns out that superoptimization is not currently feasibly, by the end of the project we will have a research roadmap describing what will need to be done next.

Superoptimization was first conceived by Henry Massalin[1], where a brute-force search was conducted through the arithmetic instructions of the Motorola 68000. The example given in the original paper is given below.

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

```
   cmp n, #0
   ble L1
   mov r0, #1
   b L3
L1:
   bge L2
   mov r0, # 1
   b L3
L2:
   mov r0, #0
L3:
```

The function on the left is compiled, giving the piece of code on the right, with multiple branches and comparisons. This could perhaps be reduced by an expert assembly programmer, but was generally accepted as close to optimal. When the superoptimizer was

run on this program, a much shorter version was found - 4 instructions long and with no branching.

		N = -4			N = 0			N = 3			
		x	d0	d1	x	d0	d1	x	d0	d1	
d0 ⟵ n		0	-4	0	0	0	0	0	1	0	
add.l	d0, d0	1	-8	0	0	0	0	0	6	0	x, d0 = d0 + d0
subx.l	d1, d1	0	-8	-1	0	0	0	0	6	0	x, d1 = d1 – d1 – x
negx.l	d0	1	8	-1	0	0	0	1	-6	0	x, d0 = 0 – d0 – x
addx.l	d1, d1	0	8	-1	0	0	0	0	-6	1	x, d1 = d1 + d1 + x
d1 ⟶ sign(n)				-1			0			1	

On the left is the instruction sequence found by the superoptimizer (Motorola 68000), with the definitions of the instructions on the far right. In the middle are three cases, computed step by step to illustrate how the code sequence works. How to instructions compute their result is not obvious exploiting the carry flag ('x') in combination with arithmetic instructions.

2 Types of superoptimizer

The most obvious choice for a superoptimizer is a brute force enumeration, simply searching all possible sequences in cost order until a correct sequence is found. However, there are other ways of performing the search, and these are also described below.

2.1 Brute force

The most common type of superoptimizer is a brute force optimizer, which enumerates each instruction sequence in turn, testing each until a correct sequence is found. The order of this search is important, since to guarantee optimality the sequences must be searched in cost order.

The search order is simple for some metrics, such as code size, where in many cases this corresponds to number of instructions in the sequences. A superoptimizer would enumerate sequences with one instruction, followed by two instructions, etc.

The disadvantage with this approach is the search space scales exponentially with the length of the instructions sequences, making long sequences intractable. The size of the instruction set also greatly affects the superoptimization potential. This is discussed below in 3.1 *Potential efficiency: Instruction set choice.* The exponential growth can be mitigated by several techniques to remove redundant instruction sequences and incorrect sequences.

2.2 Machine learning

Recently machine learning has been applied to superoptimization to traverse the search space efficiently, allowing the search to quickly identify efficient instruction sequences [2]. With this approach it is difficult to ensure optimality of the generated sequences, although the sequences are typically very close to optimal and the method gives good results in a shorter time frame than any other method.

2.3 Constraint solving

The constraint solving approach to superoptimization uses a solver (typically a SMT solver) with a set of equations to find a efficient solution. The instructions are encoded into constraints which the solver can interpret, and then the target function is represented using this form of the instructions. The SMT solver typically converts these constraints into boolean equations (a SAT problem), which is given to a SAT solver. A SAT solver then tries to work out if a solution can be found for the equations. If a solution is found then this describes the instruction sequence performs the target operation. An off-the-shelf solver can then produce a solution [3].

SMT solvers are typically very efficient, containing many heuristics to speed up and guide the search. This method allows the superoptimizer to increase in performance as the corresponding solver is developed.

However, SMT solvers typically solve a decision problem, rather than an optimization problem, meaning that the problem domain does not simply translate into SMT constraints – an iterative approach must be taken. This involves asking the solver “Prove there is no instruction sequence of size N to solve the problem”. The SMT solver will either return a counter example (the instruction sequence) or inform that the problem cannot be solved given the constraints [4]. The parameter N can then be increased until the shortest sequences is found.

3 Potential efficiency

Currently superoptimization is limited to short sequences of mostly ALU based code, with some techniques being able to handle memory operations and a few with branches. No superoptimizer has attempted to create arbitrary control flow.

Superoptimization often finds optimizations where multiple different representations of a number can be exploited. For example, if the number can be operated on as both bits and as an arithmetic number, interesting optimizations can often be found. This is also possible with floating point numbers, although no superoptimizer has attempted this yet.

One thing a superoptimizer can successfully find is branch-free replacements for short sequences of code, allowing (usually) costly branches to be removed [5]. This is focused on by many of the brute force superoptimizers. However, it is still difficult for superoptimizers to produce code with branches in them, and even more difficult if loops are considered.

3.1 Instruction set choice

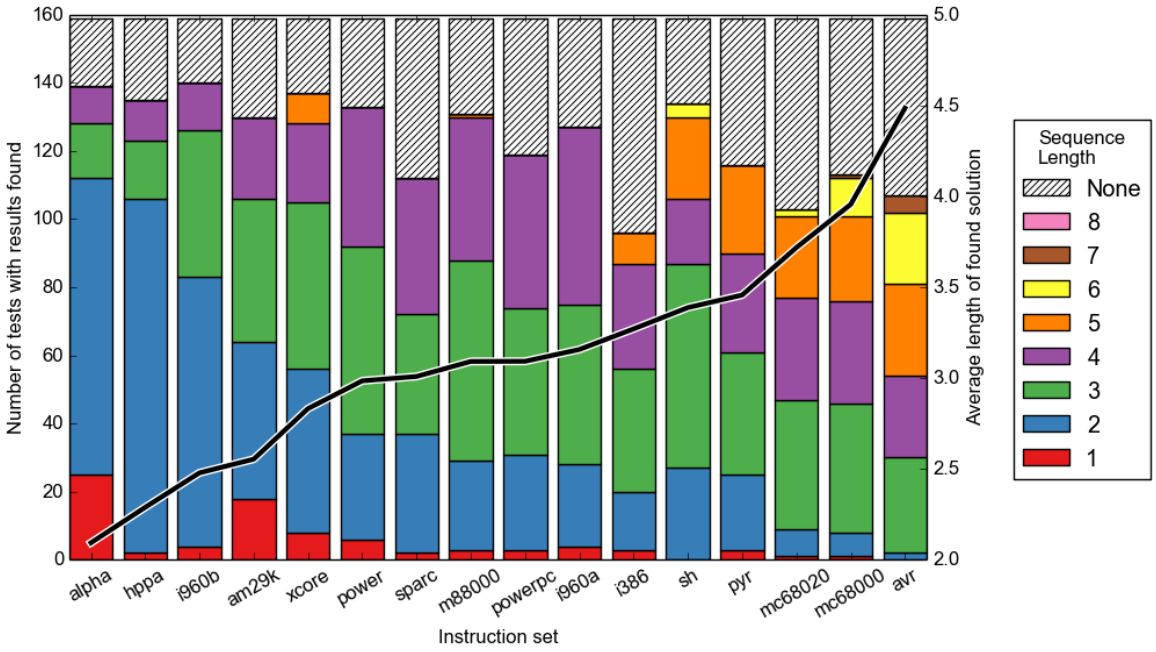
The instruction set chosen as the target for superoptimization has a large effect on the efficacy of the search – the length of the resulting solution and the time taken to find it. There are a number of trade-offs to consider when choosing the instruction set, or subset of the instruction set to superoptimize.

The number of instructions in the instruction set is the predominant factor in whether the superoptimizer will find a solution or not, since longer sequences can typically solve more complex problems. However, if each instruction only performs a 'small' amount of computation, then the required result will be need to be longer. This means that the subset of instructions chosen should be as small as possible, and consist of 'complex' operations.

The notion of operation 'complexity' is necessarily vague, since there is not a good metric to describe the amount of computation an operation requires.

The following graphs shows the superoptimization results for several architectures using the GNU superoptimizer. This includes the inbuilt superoptimization target functions (134 functions), as well as the superoptimization benchmarks given in the following section (25 functions). Each stacked bar shows the number of target functions that were able to be superoptimized, with the resulting number of instructions shown in colour.

The black line indicates the average length of the found solution for that architecture.



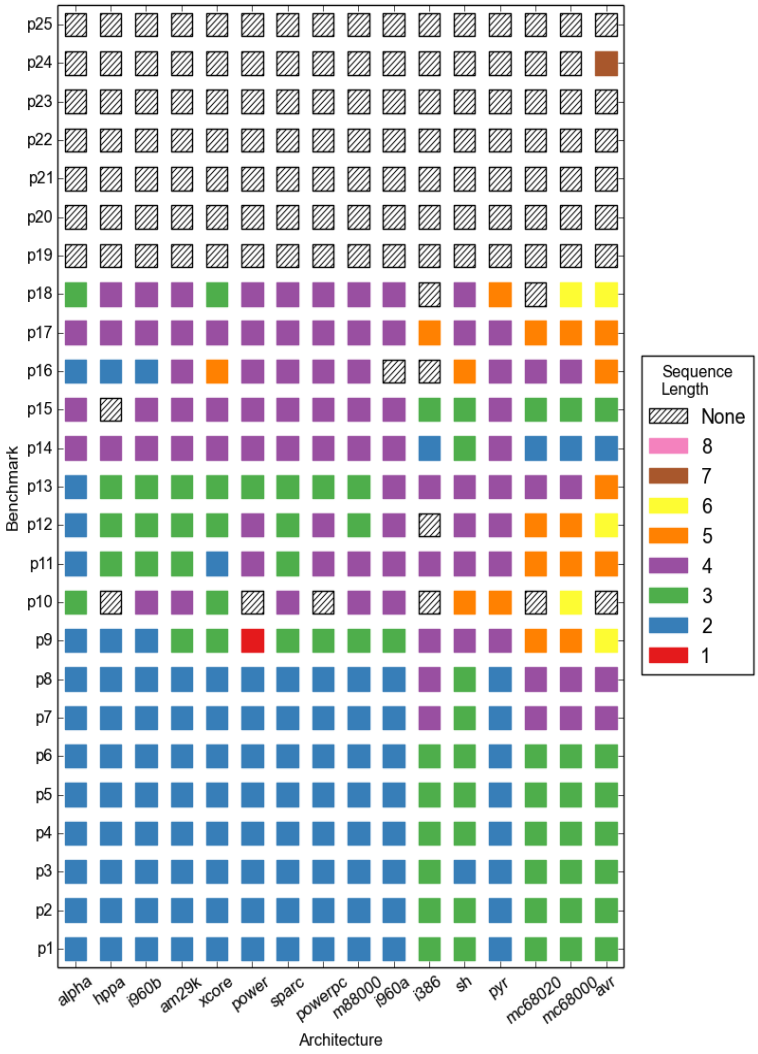
Overall this graph gives an idea of the complexity of the instruction set (or the subset implemented in GSO) for each architecture. AVR has longer instruction sequences on average, due to it being the only 8-bit architecture and by far the simplest.

An important trend to notice in the graph is that architectures on the left (lower average sequence length) typically have 3 operand instructions, whereas many on the right are 2 operand (implied destination). By not modifying one of the source registers in some operations, the superoptimizer can make reuse of this source again if necessary. If the instruction set only supports 2 operand addressing then the superoptimizer must find another way of preserving the source register, typically by inserting an additional move instruction, thus increasing the sequence size.

The average sequence length is not a measure of how good the instruction set – it is resultant from design choices made when creating the instruction set. The choice of instructions in the AVR instruction set means that the processor have a smaller silicon area than the other processors.

The importance of moving register contents around results in instruction sets with a conditional move performing well. The alpha architecture, and PA-RISC (hppa) both have some support for either conditional instructions or conditional moves.

The following graphs shows a breakdown of the superoptimizer benchmarks for each architecture. Only the 25 superoptimizer benchmarks are displayed, since it is expected these are a better indicator for the overall performance than the full set (GSO's initial functions are heavily biased towards if-conversion type optimizations). This highlights the length of the individual result for each superoptimizer trial.



As with the previous graph, there is a general increase in the number of instructions required to solve the benchmark, from left to right. From these graphs it would suggest the subset of AVR instructions implemented is the least complex, whereas the instructions from the Alpha instruction set are the most complex.

4 Difficulties

Some constructs and instructions are challenging for a superoptimizer to deal with. These are listed below.

Memory operations	Memory operations can be challenging, since it represents a large additional state that must be kept track of. This is handled in [6] by constraining the amount of memory that can be accessed to 256 bytes, by only using the last byte of the memory address. This quick test has aliasing problems in certain cases, which must be checked later with more extensive verification. Memory is also problematic in the constraint solving approach, due to

	the large number of constraints it produces – a look up table must be created with enough entries to store each unique accesses. Each entry is then compared to the lookup table when a load is performed. Volatility must also be considered when superoptimizing memory accesses.
Control flow. Branches	Branches greatly widen the scope of what the superoptimizer can generate. It is relatively simple to verify whether a section of loop free code is equivalent to another section of code with a different branching structure (i.e. allowing branching input to the superoptimizer). Generating branching output is more challenging due to the large number of possibilities.
Control flow. Loops	Loops are extremely difficult to superoptimize, particularly due to the problem of verifying loop equivalence. Superoptimizing across loop boundaries was shown to be possible in [7], however this kept the same loop structure.

Large numbers of register combinations	Many of the registers combinations are redundant, if some of the registers are orthogonal. For example:
	
	mov r0, r1
	add r1, r2, r2
	
	mov r3, r2
	add r2, r1, r1
	
	These two sequences are functionally equivalent, apart from the registers the results start and end in.

Floating point	Floating point is challenging to superoptimize because it is particularly difficult to verify if the output sequence is correct. The typical method of using an SMT solver to prove an instruction sequence's equivalence does not work for floating point (no available floating point theory).
Precoloured registers	Instructions which imply a certain source or destination registers can be problematic in certain circumstances, particularly for brute-force superoptimizers. The canonical form solution for a large number of register combinations (given above) has problems coping with this scenario, since the reduction cannot be applied to these registers. This is less a problem for converting existing sequences to canonical form but the difficulties arise when trying to iteratively generate all sequences of registers in canonical form. More research is needed to efficiently generate precoloured registers.

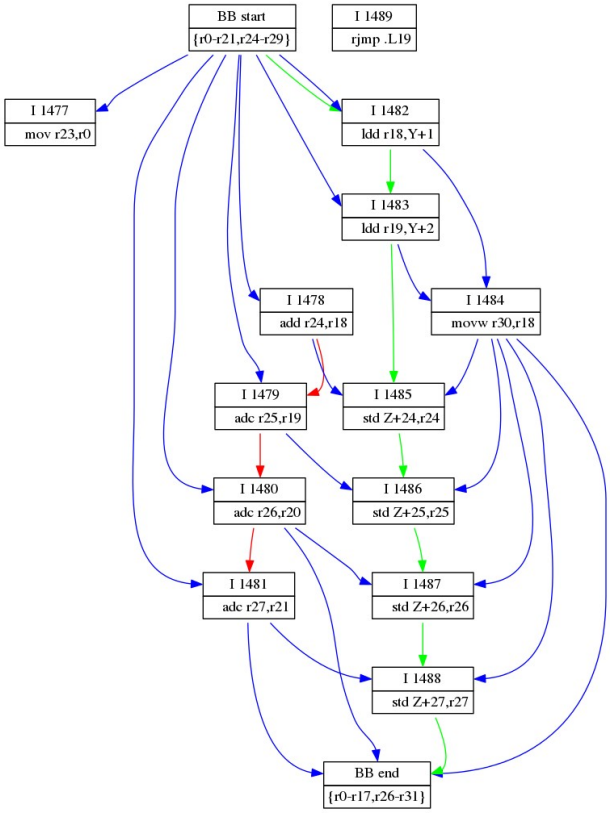
Choice of code to superoptimize

The choice of code to superoptimize has a large impact on the effectiveness of the solution, and the time taken to find it. A simple way to choose which areas of code should be superoptimized is to profile, and find which regions of code are hot. However, some of these will not be suitable for superoptimization – this section discusses some of these cases.

5 Memory accesses

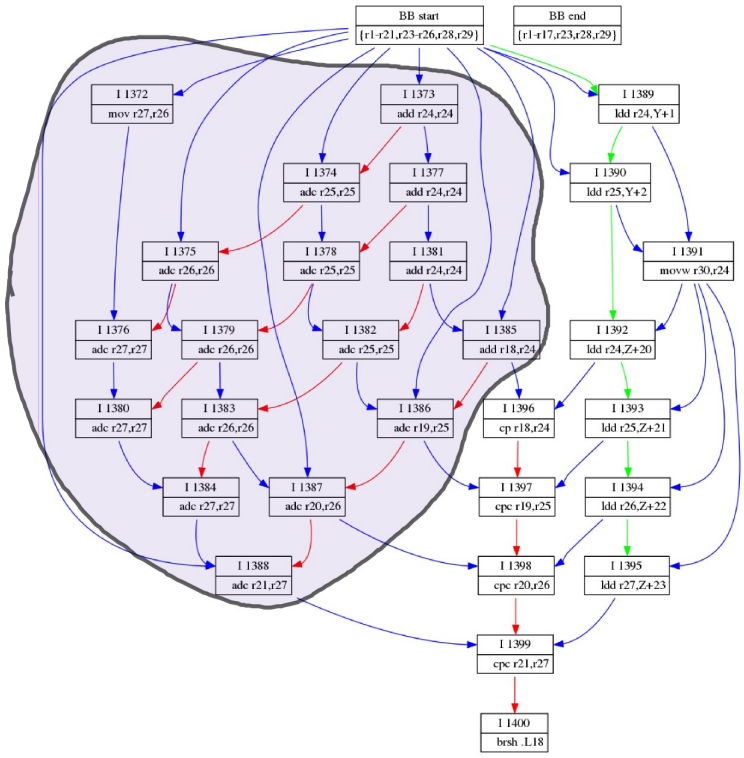
If the code frequently accesses memory then it is likely going to be difficult for a superoptimizer to utilize. Global memory accesses are often required, because it is the data the code is operating on. Local accesses frequently cannot be removed either, since they have dependencies across loop edges, or the computation of the value is a significant distance from the target region.

The following dataflow DAG shows a basic block that may not be effective to superoptimize. The green arrows indicate the sequence of memory operations. In this analysis the arrows are marked conservatively – the volatility is unknown so the order should be preserved.



If an arithmetic intensive portifc13208on of the basic block can be extracted from the rest of the basic block, then just this section could be considered for superoptimization. This has the advantage of cutting out the computation that has few possible gains, and focuses on what superoptimizers are typically most performant at.

The basic block below has a highlight region which could be fed through a superoptimizer, avoiding the chain of memory accesses on the right.



6 Target Codes

Superoptimization is not an appropriate technique for optimizing everything – it generally takes too much time to be used as a generic optimization. However for certain types of code it can be very beneficial. These are typically libraries which get used in many places, or applications which will run for a very long time and are heavily dependent on some forms of compute. In particular for embedded systems, reducing the size of a library is very important since space is heavily constrained.

libgcc	This is the runtime library that provides various support functions to applications compiled by GCC. On embedded processors which do not support floating point, this library provides the emulation, as well as other support, such as division for processors without a divide instruction. As such this library is heavily used in some embedded applications, and superoptimization could greatly benefit by reducing the overhead of this library.
compiler-rt	This library is similar to GCC, but is used with LLVM instead.
softfloat	This library emulates floating point instructions using integer and bitwise operations, for processors which do not have a hardware FPU. These functions could be ideal targets for superoptimization, since they are mostly computation bound, and not reliant on memory accesses.

Example code

Henry Massalin first introduced superoptimization with the following example [1]:

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

This function was roughly 8 instructions in length when naively compiled, but could be reduced to 6 instructions by an expert programmer. Massalin's superoptimizer managed to find the result in 4 instructions:

```
add.l   d0, d0
subx.l  d1, d1
negx.l  d0
addx.l  d1, d1
```

7 Superoptimization benchmarks

Several small functions were identified in [3] as being good targets for superoptimization. These were short functions easy to specify but had challenging implementations, many originating in Hacker's Delight [8]. The implementations of these programs often rely on interpreting the values in different representations and operating on them as such, meaning that it can be difficult for traditional compilers to solve. However, these difficulties make them tractable with a superoptimizer.

The benchmarks selected by Gulwani et al. are:

Benchmark	Parameters	Description
P1	x	Clear the least significant bit.
P2	x	Check whether x is of the form 2^n-1 .
P3	x	Isolate the right most 1-bit.
P4	x	Form a mask of set bits from the right most set bit to the least significant bit
P5	x	Set all from the least significant bit up to the least significant set bit.
P6	x	Set the least significant 0-bit.
P7	x	Isolate the right most 0-bit.
P8	x	Form a mast of set bits from the right most cleared bit to the least significant bit.
P9	x	abs(x).
P10	x, y	Test if the number of leading zeroes is the same for x and y.
P11	x, y	Test if x has fewer leading zeroes than y.
P12	x, y	Test if x has fewer or equal leading zeroes than y.
P13	x	The signum function [1].

P14	x, y	Floor of x and y.
P15	x, y	Ceil of x and y.
P16	x, y	Max of x and y.
P17	x	Turn of the right most string of 1-bits.
P18	x	Check whether x is a power of two.
P19	x, m, k	Exchange two bitfields in x, where m as a mask marking the right most field, and k is the number of bits from the start of the first bitfield to the start of the second. E.g. x = 0x00A00B00, m = 0x00000F00, k = 12 x' = 0x00B00A00
P20	x	The next higher unsigned number with the same number of set bits as x.
P21	x, a, b, c	Cycle through a, b and c, returning the next number where x = a, b or c. E.g. x = a, x' = b. E.g. x = c, x' = a.
P22	x	Compute the parity of x.
P23	x	Count the number of bits set in x.
P24	x	Round x up to the next power of 2.
P25	x, y	Compute the high part of the product of x and y.

8 Other superoptimized code

In addition to the benchmarks in the previous section, other codes have been superoptimized. Integer multiplication by a constant factor has been attempted.

Granlund et al. [5] used a superoptimizer to remove small branches and common if-then-else conditions. These were later integrated into GCC peephole pass.

Superoptimizer designs

This section discusses the design of several superoptimizers, as well as presenting a new design. These all will require varying amounts of additional research to realise (with brute-force requiring the least, and constructive requiring the most).

9 Brute force

The GNU Superoptimizer (GSO) uses a dynamic programming approach to achieve efficient superoptimization. For a sequence of length N, it will enumerate the first instructions, computing the effect on a test vector. It will then recursively enumerate sequential instructions, passing the partial computed results to each subsequent instruction. The prevents redundant calculation of the results, and quickly prunes sequences which cannot be correct.

This method of dynamic programming results in generated sequences which are in canonical form – again reducing the search space. However, the same drawbacks occur and in this framework it is difficult to implement instruction sets with implicit sources or destinations. Constraints on registers are also challenging to work with, while also ensuring that all possible sequences can be explored.

9.1 Design

A similar design to GSO could be taken, while changing the architecture slightly to enable a wider range of instructions to be easily implemented. This includes modifications to the method of selecting registers, to allow instructions with implicit destinations to be implemented, and modifications to any flag variable (currently GSO only supports the carry flag).

The superoptimizer keeps a set of current registers and flags, which are used as a quick test to exclude incorrect sequences.

Overall the superoptimizer follows a recursive design:

```
function superopt(current_len, max_len, machine_state, sequence):
    if(current_len > max_len)
        if(machine_state matches target function)
            verify the instruction sequence
        return
    for insn in instructions
        for registers in registers_lists
            execute insn with registers on machine_state
            append insn to sequence
            superopt(current_len+1, max_len, machine_state)

for i = 1 to ...
    superopt(1, i, blank_machine_state, empty_instruction_sequence)
```

This generates and tests instruction sequences in a iterative deepening depth-first search, and reuses the values computed by the instructions. This search choice is a trade-off between depth-first search and breadth-first search: depth-first is not appropriate since it does not traverse the sequences in approximate cost order. A naïve breadth-first search would use a very large amount of memory as the sequence length increased.

9.1.1 Registers

The above algorithm needs a list of all of the register variants which can go with that instruction sequence. This can be optimized by iterating through the registers in canonical form. The following algorithm will iteratively update a list of registers so that they are always

in canonical form. The algorithm is initially given a list of 0's. The length of this list is equal to the number of register slots in the instruction sequence.

```
function iterate_register_list(reg_list, num_registers)
    n = length(reg_list)
    finished = 1
    for i = 0 to n - 1
        if reg_list[i] < max(reg_list[i+1 .. n]) + 1
                  and reg_list[i] < num_registers
            reg_list[i]++
            finished = 0
            break
        else
            reg_list[i] = 0
    return reg_list, finished
```

If the algorithm returns with finished set to one, then the algorithm has wrapped around an the next instruction should be selected. The set of registers should not include slots for instructions which have an implicit register destination. For example, the AVR mul instruction has two explicit register slots, and two implicit, fixed destinations. In this case only the explicit register slots will be included in the list.

If the instruction has constraints on the set of registers that can be used in that slot, the slot should not be included in the iterative canonicalisation, and should be looped over in brute-force. This is due to the above algorithm not generating the correct results in this case (see research questions). For example, the AVR ldi instruction cannot use registers r0-r15.

9.1.2 Improvements

There are many ways to reduce the search space, however, the reduction in search space is a trade off with their implementation efficiency.

Commutativity	Some instructions have commutative operations. These can be exploited to reduce the search space by only considering one form of the instruction. Commutativity also interacts with canonical form, meaning that the ordering of the register slots in can be important.
Redundant outputs	This technique involves analyzing the generated instruction sequence for dependencies that would have been eliminated in fully optimized code. The number of registers in use, and used by the input sequence is known, therefore if the sequence reads from a register that has not been assigned to yet, it can be excluded. Writes to registers which are never read from can be removed. These can be incorporated into the canonical register generator, by giving it information
Machine state lookup	This exploits the fact that the majority of instruction sequences are incorrect. The machine state is stored in a hash table, along with the amount of instructions (or cost) required to compute it. Each time the superoptimizer is about to recurse, it first checks whether the current machine state is in the table. If it is, and the cost to reach that state is higher than that stored in the table, then the resulting sequence cannot possibly be optimal (since we've previously seen that machine state, and not found a result).

	This leads to 3-10x speed up in certain circumstances. This method needs tuning, since the hashtable lookup can easily slow the search down. In particular, it is not beneficial to store or lookup the machine state if there is only 1 instruction left in the current search. This method essentially removes parts of the search space which are known to be bad, typically occurring when some instruction reordering does not affect the output of the sequence. This also has the effect of quickly pruning sets of sequences which have dead code in them. Sometimes trivial sequences of instructions produce the same output (e.g. `add r0,-1` and `sub r0,1`) and if these are known to be incorrect then are excluded by this technique.
Heuristics	Some heuristics can be used to change the order of the iterated instructions. This does not reduce the search space, but may allow the superoptimizer to find any solution quicker.

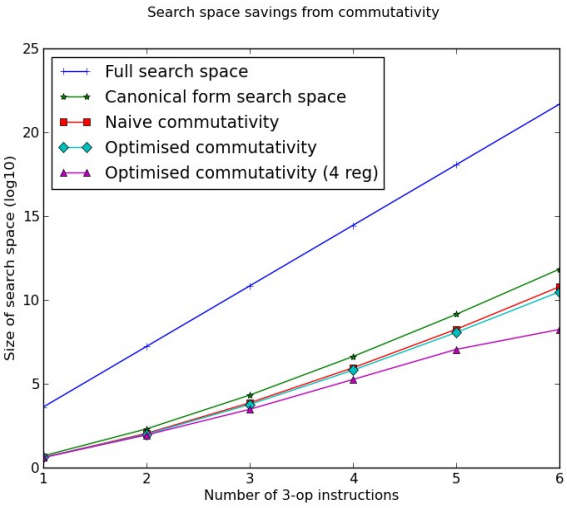


Illustration 2: Savings from implementing canonical form, and commutativity

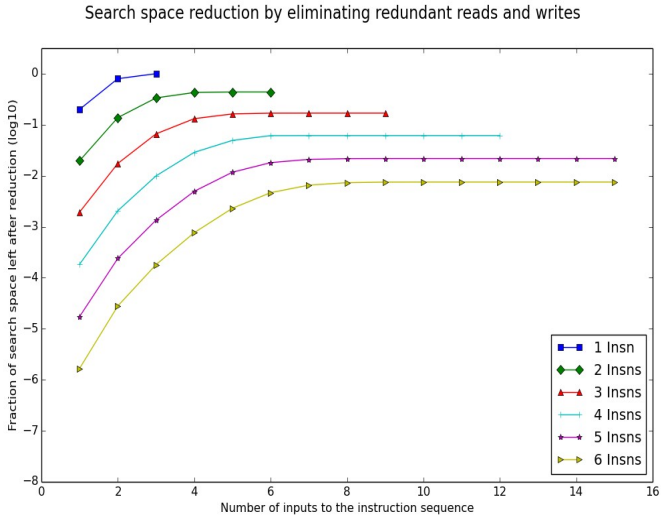
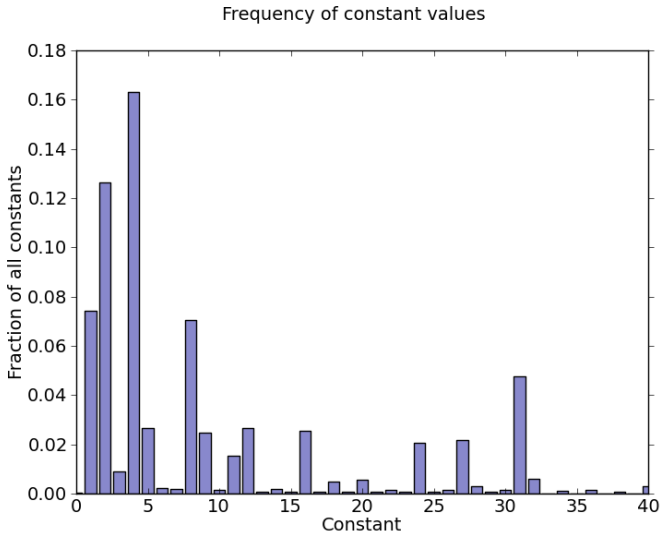


Illustration 1: Search space savings from not generating code with redundant read and writes

9.1.3 Speed or Optimality

There is a certain amount of cross over with the machine learning approach if heuristics are used to guide the search. In particular a brute force superoptimize has difficulty with the possible range of constants – there are frequently too many possible constants to use them all. GSO only uses a small range of constants, removing the guarantee of optimality but speeding up the search. The set of constants is chosen carefully, based on constants which appear frequently in code. The graph below shows the frequency of certain constants for a range of benchmarks.



The graph shows spikes around the powers-of-two, and powers-of-two minus 1. The superoptimizer could bias the search by preferring power-of-two constants, and then trying other constants after.

9.2 Advantages

- Simple to implement and a well known technique.
- Guarantee of optimal solution, provided the exploration order is sound.
- Can be parallelized easily.

9.3 Disadvantages

- Restricted to small sequences of code.
- Choosing the exact subset of the instruction set to implement in the superoptimizer, or the subset for superoptimization is hard.

9.4 Research Questions

- How can sequences be enumerated in cost order when the machine model is complex? This could require an amount of "fuzziness", where when a solution is found, the superoptimizer continues for a certain amount of extra cost.
- Can we build a fast way of recognizing areas of the search space which are never optimal, and quickly exclude them?
- Can the canonical form of registers be extended to cope with instructions that have constraints on the register slot?
- The machine state lookup improvement is heavily dependent on the initial values chosen for the registers. Is there a way of choosing a good set of values?
- What other techniques are there of improving a brute-force search?

10 Learning Optimizer

STOKE [9] is an optimizer based on machine learning techniques to guide the search for an instruction sequence. This type of superoptimizer is able to find much longer code sequences in the same amount of time (up to 16 instructions).

A similar design could be embedded into a compiler, since the search can utilize an existing sequence as the starting point for the superoptimization. Then the sequence can be permuted until a better sequence is found.

10.1 Design

A compiler pass, placed near the end of the compilation process that will use machine learning techniques to attempt to improve the code. Initially the pass would work on individual basic block, selected by a heuristic based on the length and type of instructions in the basic block.

The chosen machine learning method is then applied to the instruction sequence (see the next section), and this sequence is tested (requiring the compiler pass to be able to simulate instructions for the target machine). If the sequence matches then it is verified if it is correct, and a cost model applied. If the sequence is correct, and performs better then either the search can stop there, integrating that sequence into the code, or the search can continue and attempt to find better sequences. Otherwise the search continues until the allotted time has expired.

10.2 Machine Learning

The most challenging part of this approach to superoptimization is choosing a fitness function that guides the search towards a correct sequence. STOKE used a function which measured the number of correct bits in the output registers, combined with the performance model to guide it towards efficient solutions. This is necessary for creating solutions from scratch, and if the search accepts an invalid sequence.

STOKE used a Monte Carlo Markov chain method (similar to simulated annealing) to choose which sequence to use next. This approach mutated the instruction sequence, then accepted it if the fitness of the solution had gone up. The solution was also accepted if it was less fit, with a certain probability dependent on the value of the fitness.

Other learning schemes could be used, such as genetic programming or genetic algorithms. The performance model can easily be swapped out for other metrics, such as code size and energy consumption.

10.3 Advantages

- Able to explore longer instruction sequences.
- Reduced time to find a solution compared to other methods.
- Giving the compiler a longer time to perform the optimizations will generally result in better solutions.
- Some machine learning techniques can be parallelized easily, allowing faster exploration.

10.4 Disadvantages

- No guarantee of optimality.
- There can be many parameters to tune, which greatly affect the time and solution found by the optimizer.

10.5 Research Questions

- If the code was modified since the last compilation, can the previous results of the search be used as an initial seed, and guided towards a correct sequence.
- Which method of machine learning works well on this kind of problem?

- How close to optimality does this approach come? How long does the superoptimizer have to be run to get within X% of optimal?

11 Constructive

This section presents the experimental design for a constructive superoptimizer. The premise of the superoptimizer is to split the synthesis task into two components – creating an optimal dataflow DAG, then optimally lowering this DAG to sequential instructions.

This approach allows unsupported instruction sequences to be superoptimized, by superoptimizing 'around' them.

11.1 DAG superoptimizer

This portion of the superoptimizer takes a sequence of code as input, and produces an optimal DAG as output.

1. Compile the code at the desired optimization level.
2. Construct a control flow graph from the code, and perform liveness analysis on the code.
3. For the targeted sections of code, construct a dataflow DAG for each basic block.
4. Simplify the DAG, so that register assignments are removed, and implied by the DAG.
5. Compose the multiple basic blocks into a large DAG.
6. Transform the DAG into SMT constraints – this is used for the verify step.
7. Construct the skeleton DAG.
8. Iteratively perform Counter Example Guided Inductive Synthesis to fill in the holes in the DAG.

11.1.1 The dataflow DAG

The dataflow DAG is constructed per basic block, and describes the dataflow within that basic block. This is constructed by placing an edge between the instruction that generates a value, and an instruction that uses that value. If no instruction in that block creates the value that is needed by an instruction then the register holding that instruction must be live on entry to the block. The liveness analysis provides a set of registers which are live on exit from the block. If an instruction creates a value which is not used in the basic block, the registers which are live on exit from the basic block indicate whether the instructions output should be on an edge out of the block.

Once the dataflow DAG has been constructed, the register names are removed. All of the memory operations are represented sequentially unless volatility can be ignored. If volatility is to be ignored, alias analysis can be performed to break the dependencies.

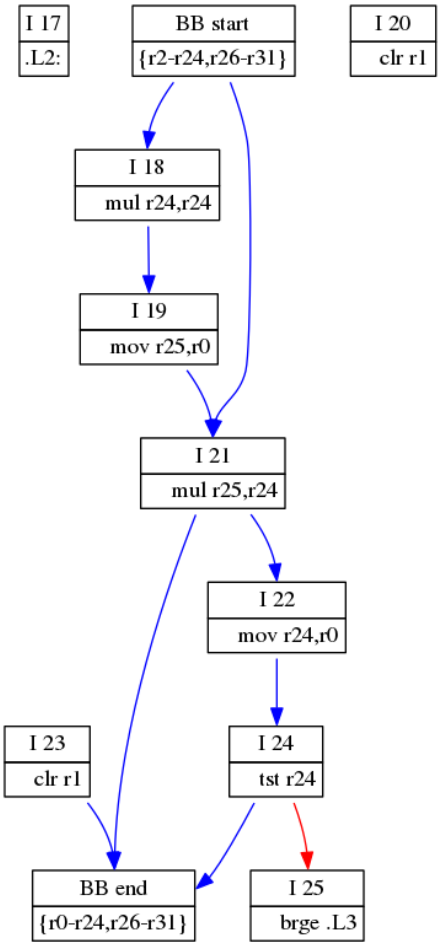
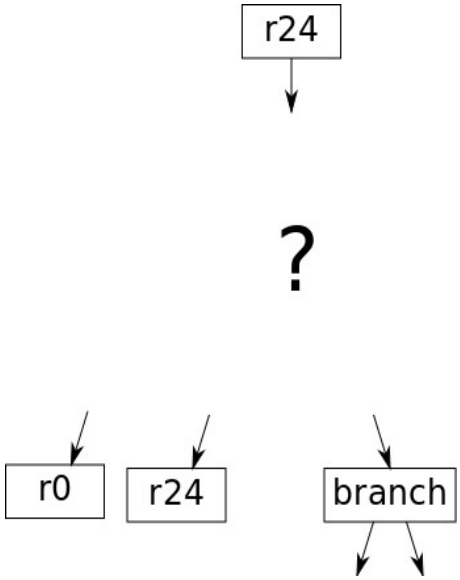
11.1.2 DAG composition

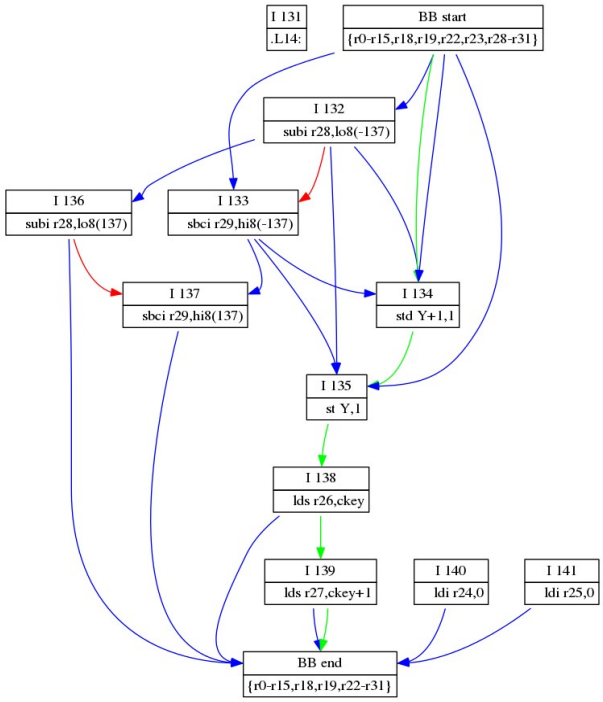
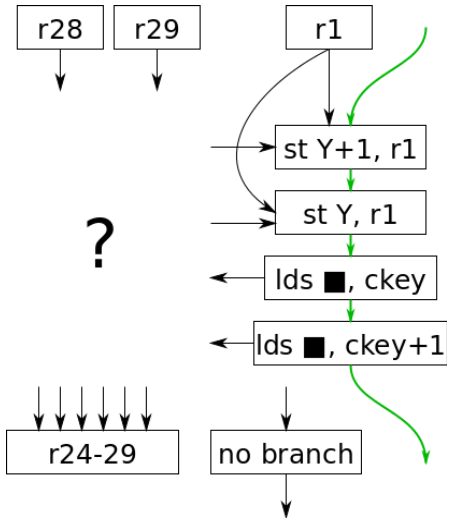
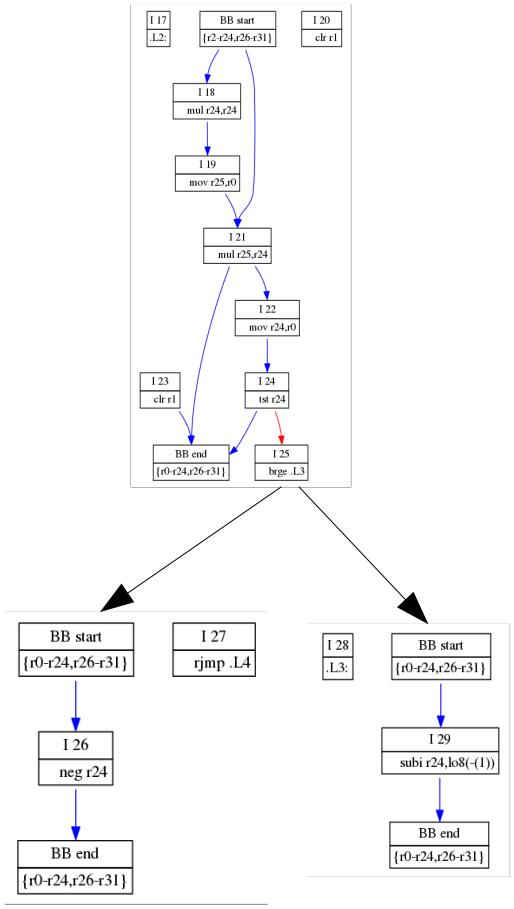
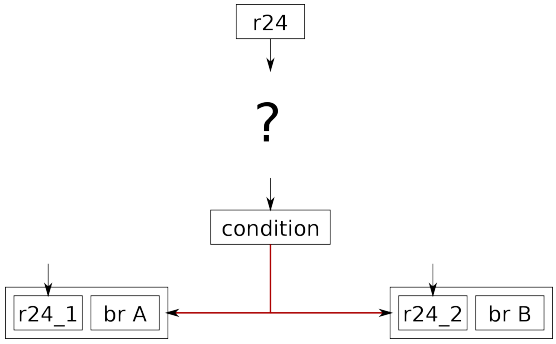
Multiple DAGs can be composed together, by linking the registers in to each DAG with the registers generated by the previous basic blocks. In the case where there are multiple previous basic blocks there would be multiple edges into a node for a single value. These should be intermediary multiplexor (phi) nodes inserted. These multiplexors choose the values to propagate, based on the selector's value. The selector's value is chosen based on conditions generated in previous basic blocks.

11.1.3 Skeleton DAG

The skeleton dataflow DAG is a reduced form of the previously constructed DAG, with only the portions which are difficult to superoptimize remaining. For example, if memory

operations were not implemented in the superoptimizer, the memory operations and their connections would remain in this DAG. The following diagrams show examples of the skeleton DAG. The blue arrows indicate a register dependency, red indicates a dependency on a flag and green indicates a memory dependency.

Dataflow DAG	Skeleton DAG
	 This dataflow DAG only operates on r24 (r0 is used by the multiply, r1 is cleared after the multiply). Therefore only r24 needs to be considered for input and output, along with the flag dependency, indicating to the branch.

Dataflow DAG	Skeleton DAG
	 In this example, the chain of memory accesses is kept identical, with the dependencies. The rest of the instruction can be filled in using the values provided by the existing instructions.
	 This example indicates how multiple basic blocks can be combined into a single skeleton DAG. Two outputs for r24 must be produced, along with a condition that selects between them. This is selected by a multiplexor component in the DAG. The “br A” and “br B” indicate the next destination of the code is also dependent on the flags. This form of the DAG allows the potential for control flow to be encoded, then subsequently split into separate basic blocks or a conditional move performed (as decided by the lowering step).

11.1.4 Filling in the skeleton DAG

Both the dataflow DAG and the skeleton DAG are converted into SMT constraints. The corresponding points on the graph can then be asserted for equality – the inputs, the outputs, and the intermediary nodes in the skeleton DAG.

Counter Example Guided Inductive Synthesis can then be performed, with the set of instructions in the instruction set as components. With an interactive increase in the number of components available, this will result in a optimal solution to make the skeleton DAG and dataflow DAG equivalent.

11.2 Lowering superoptimizer

This takes in the optimal DAG as input, and outputs a sequence of assembly level instructions. This can be performed by using integer linear programming (ILP) to represent the ordering of the instructions and perform optimal register allocation. Additionally this can optimize the splitting of the computed DAG into multiple basic blocks. By using an ILP solver, the metric for superoptimization can be considered:

- Code size: minimize the control flow such that the total code size is small. This will likely result in code which performs slowly, as if conversion may be done on the entire set of blocks.
- Performance: minimize the average path length when the control flow is included. This could use profile data on the range of values the multiplexor expressions take, allowing control flow which minimizes the execution time to be inserted.

More research is needed for the exact formulation of the ILP problem (it is covered in detail in [10]–[12]), however, the DAG itself can be lowered using traditional compiler techniques (register allocations, scheduling).

11.3 Advantages

- Decomposes superoptimization into smaller more manageable problems. This allows the compiler to substitute in non-optimal techniques if a particular subproblem is taking too long.
- Able to generate multiple basic blocks.
- Possibly able to handle large sequences of code (20 instructions).

11.4 Disadvantages

- Complex to implement.
- Currently untested and may not provide the speed ups necessary.
- Difficult to parallelism (few efficient parallel SAT solvers exist).
- Decomposition of the problem may result in no guarantee of optimality.

11.5 Research Questions

There are several additional items to be researched before this design can be realized.

- Does a minimal DAG correspond to a minimal instruction sequence? Can the DAG synthesis be biased towards DAGs which make the ILP problem easier?
- What is the best way of formulating the ILP problem?

Conclusion

Today there are only a handful of areas where superoptimization can be applied effectively in a commercial context. In particular, there are significant limitations on producing code sequences which include memory and branching operations. This paper has shown how the commercial value of superoptimization can be increased in both long and short term.

Three different designs for superoptimizers were analyzed: those based on brute-force, those applying machine learning; and those using constructive approaches. Overall, the brute-force superoptimizer has the lowest risk, being built on well known technologies. However, it has the most limitations, being restricted to a small number of instructions. On the other end of the scale, a superoptimizer taking a constructive approach with SMT solvers may be able to scale to much longer sequences, however the technology is in its infancy and needs extensive development. The machine learning based superoptimizer has the most potential for immediate benefits, with a smaller amount of additional research to develop, and potentially large gains. This superoptimizer may not be able to guarantee an optimal solution is found, but can quickly explore possible solutions and improve code size/speed/energy.

The features of the instruction set greatly affect the efficacy of superoptimization and there are a number of instruction set features which cannot be superoptimized. Some of these are fundamentally difficult, such as a superoptimizer producing loops, while some are implementation difficulties, such as supporting memory accesses.

As a result we can see three stages to the future deployment of superoptimization.

1. Immediate use of the GNU superoptimizer (a brute force tool) to help customers optimize key sections of code, with short-term improvements to make it easy to generate goal functions automatically from source assembler. This is quite a niche application, and its commercial value is largely in cementing Embecosm's technical credibility.
2. In the medium term, application of DAG analysis to increase the applicability of the brute force approach and development of machine learning technology to make the brute force approach more directed. This is still relatively high risk, and would be appropriate for feasibility funding by Innovate UK.
3. In the longer term, investment in constructive approaches, with academic study to determine how to handle loop generation, memory access and floating point. The starting point for this should research council funded academic projects, but in which Embecosm would provide some industrial support.

However this study has identified some areas with scope to increase the commercial relevance with some relatively modest R&D investment, albeit at quite high risk. There is much greater scope longer term, but this will in the first instance require major academic investment to resolve issues with memory access, branching and floating point.

This has been a very high risk study. Embecosm would like to express our thanks to Innovate UK for their support, without which we could not have carried out this work.

References

[1] H. Massalin, “Superoptimizer - A Look at the Smallest Program,” *ACM SIGARCH Comput. Archit. News*, pp. 122–126, 1987.

[2] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic optimization of floating-point programs with tunable precision,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI ’14*, 2013, pp. 53–64.

[3] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of loop-free programs,” *ACM SIGPLAN Not.*, vol. 47, no. 6, p. 62, Aug. 2012.

[4] R. Joshi, G. Nelson, and K. Randall, “Denali : A Goal-directed Superoptimizer.”

[5] T. Granlund and R. 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*, 1992, pp. 341–352.

[6] S. Bansal and A. Aiken, “Automatic generation of peephole superoptimizers,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, p. 394, Oct. 2006.

[7] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, “Data-driven equivalence checking,” in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications - OOPSLA ’13*, 2013, vol. 48, no. 10, pp. 391–406.

[8] H. S. Warren, *Hacker’s Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *Architectural Support for Programming Languages and Operating Systems*, 2013, p. 305.

[10] D. W. Goodwin and K. D. Wilken, “Optimal and Near-optimal Global Register Allocation Using 0- 1 Integer Programming,” vol. 26, no. January 1995, pp. 929–965, 1996.

[11] T. C. Wilson, G. W. Grewal, and D. K. Banerji, “An ILP Solution for Simultaneous Scheduling , Allocation , and Binding in Multiple Block Synthesis,” 1994.

[12] D. Kastner and M. Langenbach, “Code Optimization by Integer Linear Programming,” in *Compiler Construction*, 1999, pp. 122–137.