Symbolic Superoptimization

Jonathan Hill

October 14, 1999

Abstract

State-of-the-art ad-hoc methods of code optimization, although highly effective, have been shown to produce locally sub-optimal code. This has been demonstrated by superoptimization, the process of brute-force search for an optimal code sequence. Superoptimization is currently far too expensive to be applied in a traditional optimization framework. However, it may be of particular use in optimizing frequently executed code such as loop kernels. The extent of superoptimization's usefulness depends on the extent to which it can be made more efficient. In this report, we determine greater efficiency through reduction of the superoptimization search space. Our methodology is to locate machine independent mechanisms facilitating this reduction. These mechanism will compute on semantic information about the machine resources and instruction set. We investigate and determine the limitations of a lower-than-instruction, bit-level, semantic. This contrasts with the results we obtain from a higher-than-instruction, functional-level, semantic. The later generates a more efficient search space than the standard instruction-level semantic, and can be automatically generated from an instruction-level semantic at compile-compile time. This document also explicates the infrastructure developed and utilized to obtain these results.

1 Introduction

Superoptimization is the strategy of searching all possible programs, from length 1 to n, for a program which performs a desired function. This method of code generation is simple and general-always producing optimal code. It can only achieve limited use, however, because the size of the search space with program length grows exponentially. In particular, for a meaningful program space, the constants tied to the rate of growth of the search space limit searches to lengths of 7 or less instructions.

In an attempt to tackle the search-space problem we have developed a symbolic superoptimizer. Our superoptimizer is machine independent, with the instructions to combine in the search space represented by bit-level Boolean algebra. More importantly, our representation of instructions allows us to maintain complete static and run-time information about the programs generated in the search space. Our goal was to utilize this semantic information to reduce the size of the search space, and therefore extend the length of achievable optimal programs. We have developed a compiler-compilation time method for the reduction of the search space to functional units.

This document is divided into five main sections. In the first, some background on superoptimization in provided. The second section details our investigation of methods of superoptimization using bit level semantics. The third section considers the representation of superoptimization using functional-level semantics. The fourth draws conclusions on our investigation and discussed future work. The fifth section consists of appendices with an overview documentation for our current superoptimizer infrastructure.

2 Background

Exhaustive search as a technique for code generation was suggested and implemented by Krumme and Ackley[?] Their code generator utilized a table-driven approach, in which reasonable alternatives of instructions were listed explicitly. An expression tree was translated to machine instructions by covering the tree with the most effective combination of instructions from the tables.

Davidson and Fraser [?] provided that machine descriptions can be utilized to identify when groups of instructions can be reduced to machine instructions during code generation. They and Kessler [?] implemented classical peephole optimizers that utilized patterns discovered by exhaustive combination searching at compile-compilation time.

Massalin [?] noted that many optimal instruction sequences to perform functions did not resemble those generated by tree-covering expression matching code generators, before or after advanced peephole optimization. He implemented the first superoptimizer, which searched all possible programs to length n (without branching instructions or memory references) which produced a desired expression (function.) Massalin developed a symbolic model, and finding this to be too slow, a machine-dependent dynamic code-testing model. His results with the later implementation suggested programs of 7 or 8 machine instructions could be generated within reasonable time (an hour) when seeking truly optimal code for a function. Furthermore, the resulting code was often surprisingly elegant and efficient, not to mention unusual in appearance.

Our goal was to extend the approach of Massalin's symbolic superoptimization, by seeking methods of pruning the search space of programs in the exhaustive search. A heuristic method for pruning the search space rate of growth would allow superoptimization to be applied as a practical local code optimization strategy. This would be an extension of the more local instruction combination search of the PO optimizer by Davidson and Fraser [?]. At the other extreme, an effective depth first search algorithm would render all other code generation methods obsolete.

3 Motivation

It is the opinion of the author that do to the 'triviality' of the analysis many author's see in the search space size calculation, that it has not been well documented. We now explicate the size of the search space for a superoptimizer.

Given a instructions, each of which has b parameters, such that each parameter has c options, in a program of length n, then there are $a * b * c$ possible instructions to fill each of the n slots of the program.

> Actually there are somewhat less, since not all instructions have the same number of operands.

The number of total possible programs of length n is therefore

$$T_n = (abc)^n \tag{1}$$

> Again this is a simplification, since the number of operand possiblities (c) varies with data flow for a particular program fragment.

The size of the search space for all programs up to length m is

$$S = \sum_{n=1}^{m} T_n = \sum_{n=1}^{m} (abc)^n = \frac{1 - (abc)^{m+1}}{1 - (abc)} \tag{2}$$

This result approximates, for large (abc) to

$$S = (abc)^n \tag{3}$$

The larger the constant (abc), the more the search space is dominated by its final level. This is true of any exponentially growing search space.

What is important to note for superoptimization, is *the large magnitude of the constant* (abc) for reasonable instruction sets. For example, with just 12 register to register instructions, and 5 general purpose registers, $(abc) = 120$. The following table summarizes the size of each level of the search space, and the time taken to search it with $500,000$ programs per second searched (as would be capable of Massalin's superoptimizer on today's computers.)

n	program options	time to search
1	120	.24 ms
2	14,400	30 ms
3	1,728,400	3.46 s
4	207,360,000	6.91 minutes
5	24,883,200,000	13.8 hours
6	3,185,049,600,000	9.8 weeks
7	382,205,952,000,000	24.31 years

In reality, we can prune this search space so that the number of register options grows only with instruction position. We can also eliminate certain do-nothing instruction sequences. However, many instructions that may seem to be of little value cannot be eliminated, as they show up in superoptimal programs.

Thus, the search space can be pruned slightly by this add-hoc technique. However, there is a danger in add-hoc pruning. Since useful instruction sequences may possibly result from any combination of instructions that have any effect, we should be wary of removing instruction sequences whose effect can not be generated in some other allowed form.

One can see that for a reasonable instruction set size, we quickly cross the boundary of reasonable superoptimization at a number of instructions in the first order of magnitude. Our goal was therefore to reduce significantly the base constant in the geometric series determining the size of the search space. It was reasoned that bit-level data-flow information carried the essential semantics of superoptimal code fragments. There for we would attempt to prune the search space through use of bit-level semantic information.

To this end, an assembly instruction description language was developed. This language is called BTL. BTL is designed to model instructions in the form of guarded assignments on Boolean expressions. A low-level algebra was chosen over a higher-level representation of bit level semantics do to the purely Boolean algebraic relationship between a superoptimal program and a more human form of the same program.

4 Bit Level Semantics for Search Space Pruning

In order to significantly prune the search space, we would need a way to evaluate which program fragments are likely to yield short programs performing the desired functions.

Consider a program fragment P_2. If this fragment is a valid ending to a program that will perform function F, then there is some program P_1 such that $Semantic(P_1 P_2) = F$.

Since we are seeking short programs, and given P_2, we ask if

$$\exists_{P_1} : Semantic(P_1 P_2) = F \;\; length(P_1) + length(P_2) <= N \tag{4}$$

A good heuristic for depth first search would provide a function $H(P, l)$ which given program fragment P would return the probability of the function completing by $length(l)$.

We now discuss the models which were applied to attempt to derive such a function.

4.0.1 Precondition Model

The hypothesis for a precondition modeling superoptimizer is that given a program fragment P_2, $preconditions(P_2)$ would give us information for the calculation of an H function. This did not turn out to be the case.

Given that we have the postcondition for the function, F, we backwards substitute the bit-level semantics of the instruction instances into the postconditions of F to determine the preconditions of P_2. The resulting weakest preconditions are then tested to see if they cannot be valid by some combination of data, in which case the program fragment has been shown to be invalid as an ending sequence to implement F. Unfortunately, few P_2's can be shown to be invalid in this way. The only invalid cases are instruction sequences which irreversibly throw away input information. These could easily be shown to be invalid by higher level, less costly analysis.

The inherent failure of weakest preconditions to prune the search space lies in the 'weakness' thereof. A precondition of a program fragment merely states that the current contents of some registers, in some Boolean expression, must somehow contain an arrangement of bits which satisfies F. The weakness of this statement means that regardless of the form of the weakest precondition, so long as it does not limit the input, some yet to be analyzed previous code fragment P_1 could exist to satisfy the precondition required for P_2.

Consider the example of the very simple program

- AND D0, D1

Modeled on two bit registers on the motoralla 68k instruction set.

The resulting postconditions, F, for this program are:

$$D1_1[i] == (i = 0..1)(D1_0[i] * D0_0[i]) \tag{5}$$

Now consider we are looking at the program fragment, P_2 defined as

- OR D0, D1

$$D1_1[i] == (i = 0..1)(D1_0[i] + D0_0[i]) \tag{6}$$

For this program fragment, the resulting precondition that would generate F for a program ending with P_2 (an OR D0,D1 statement) is

$$(D0_1[i] + D1_1[i]) == (i = 0..1)(D1_0[i] * D0_0[i]) \tag{7}$$

Lets examine this precondition. It states that in order for *OR D0,D1* to appear as the last statement in a program equivalent to F, the *OR* of the bits in D0 and D1 before the OR instruction must be the same as the *AND* of the bits in original input of D0 and D1. In other words, some P_1 must exist that takes the original contents of D0 and D1 and modifies them in such a way that ORing them will produce the output for F.

Such weak preconditions make it difficult to rule out that such a P_1 might not exist. Consider for the case of registers having only 1 bit. Then, we can describe F in terms of D0 and D1 as follows:

D0	D1	F=D0 & D1
1	1	1
1	0	0
0	1	0
0	0	0

What P_1 functions would map the contents of D0 and D1 so that it would produce the required preconditions of P_2? We could find functions by searching all P_1 programs for such a mapping, but this is precisely the brute force search of superoptimization that we sought to avoid!

Perhaps there is an alternative method to suggest the existence of a P_1 of $length(P_1) <= l$? Let us consider not enumerating all programs P_1, but instead enumerating all short Boolean functions that would satisfy the preconditions. If such Boolean functions are found, we would then have to search programs P_1 to find a match to one of our Boolean functions. But if no such Boolean function exists, we can eliminate this P_2 from our search.

Continuing with the 1 bit register case, there are three mappings for semantic(P_1) which will suffice. These result in three programs P_1 which satisfy P_2. In table form these are:

-
$$\begin{aligned} D0_1 &= D0_0 D1_0 \\ D1_1 &= 0 \end{aligned} \tag{8}$$

$D0_0$	$D1_0$	F	$D0_1$	$D1_1$	$D0_1 + D1_1$
1	1	1	1	0	1
1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	0	0	0

-
$$\begin{aligned} D0_1 &= 0 \\ D1_1 &= D0_0 D1_0 \end{aligned} \tag{9}$$

$D0_0$	$D1_0$	F	$D0_1$	$D1_1$	$D0_1 + D1_1$
1	1	1	0	1	1
1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	0	0	0

-
$$\begin{aligned} D0_1 &= D0_0 D1_0 \\ D1_1 &= D0_0 D1_1 \end{aligned} \tag{10}$$

$D0_0$	$D1_0$	F	$D0_1$	$D1_1$	$D0_1 + D1_1$
1	1	1	1	1	1
1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	0	0	0

Does a program fragment P_1 exist to match one of these $semantic(P_1)$? Yes, the following program P_1 matches the third form:

- MV D1, D2
- AND D0, D1
- AND D2, D0

There are additional sequences, two programs for F that can be produced from a final OR instruction are:

$$AND\ D0, D1 = \begin{array}{r} MV\ D1, D2 \\ AND\ D0, D1 \\ AND\ D2, D0 \\ OR\ D0, D1 \end{array} \tag{11}$$

and

$$AND\ D0, D1 = \begin{array}{r} AND\ D0, D1 \\ MV\ 0, D0 \\ OR\ D0, D1 \end{array} \tag{12}$$

While such programs for AND are clearly not desired, precondition calculation alone cannot disqualify an OR instruction as a valid choice for P_2. Additional analysis would be needed. Such analysis would require determination of the minimum length of the fragment P_1 required to satisfy the preconditions of P_2. We have been unable to determine an analysis technique, apart from brute force searching of programs P_1, that could make such a determination. However, this technique exemplifies the advantage of another pruning technique explored in the next section of this document.

Comparison with Simulation A weakest precondition based search algorithm tests the preconditions of program fragments to see if they are satisfied for all inputs, in which case we have discovered that $semantics(P_2) = F$. This is the successful termination condition of the search algorithm.

Since SAT is NP, such tests for validity and successful matching are expensive in the number of bits and instructions of the program. Furthermore, the length of the Boolean form of the instructions tends to grow exponentially with the number of bits of the instructions. Thus, only limited bit-word models of instructions are valid. (Our experiments utilized 2, 3, 4, and 5 bit words.)

Precondition calculation involves full forward propagation of the reference program to generate F. Backwards propagation, generating preconditions on a P_2, is in itself a slow process if we wish to reduce the Boolean equations to minimal form. Failing to do such reduction results in their exponential growth, so that determination of preconditions is an exponential growth problem in the length of the program.

For these reasons, precondition-form superoptimization is very inefficient compared to simulation of instructions. In practice precondition form superoptimization searches several order of magnitude slower than simulation. Much of this discrepency is do to the large amount of computation involved in backward substitution, involved in Boolean expression simplification. This coupled with out object oriented approach and heavy reliance on dynamic objects, further compunds the speed discrepency.

4.0.2 Output Correlation on Simulated Instructions

In our second model, we attempted to use data output of a P_1 to determine if a P_2 was likely to exist in length N. Given a P_1 and F, for each data input combination, the correlation between the output of P_1 and F was taken. It was hypothesized that perhaps more fit P_1 programs would exhibit higher output correlation than programs P_1 that could not generate function F within a short number of instructions. This hypothesis was contradicted by our data.

A 5 bit model was used to seek a correlation matching for the absolute value function, which can be written as

```
        MV D0,D1
        CMP 0,D0
        BGE ENDER
        MV 0,D1
        SUB D0,D1
ENDER:  NOP
```

The following program fragments had maximal correlation for program length 3:

```
        ADD D1,D1
        SUB D0,D1
        AND D0,D1
Correlation: 0.833333 on D1
        SUB D1,D1
        SUB D0,D1
        AND D0,D1
Correlation: 0.833333 on D1
        MV D0,D1
        NEG D0,
        AND D0,D1
Correlation: 0.833333 on D1
        MV D0,D1
        NEG D1,
        AND D0,D1
Correlation: 0.833333 on D1
        MV D0,D2
        NEG D0,
        AND D0,D2
Correlation: 0.833333 on D2
        MV D0,D2
        NEG D2,
        AND D0,D2
Correlation: 0.833333 on D2
        EOR D1,D1
        SUB D0,D1
```

```
        AND DO,D1
Correlation: 0.833333 on D1
```

At length 5, we can achieve near correlation with

```
        ADD  D1,D1
        ADD  DO,DO
        SUBX DO,D1
        ADDX D1,D1
        ADDX DO,DO
Corelation: 0.916667 on DO
```

and other variants.

The actual superoptimal code fragment of length 5 is

```
      MV   DO, D1
      ADD  D1, D1
      SUBX D1, D1
      EOR  D1, D1
      SUB  D1, DO
```

Interestingly this data suggests that we can find 'near' programs that *almost* work. They tend to perform F for a majority of input cases but not all. This might have some application to efficient fuzzy computation, where we want to be right most of the time, but not all the time. However, the distribution of input values over which the *almost* Fs fail tend to be clustered. There seems to be little predictive use of output correlation that could be used to prune the program search space.

*N*ote: Coupled with our results for search space size(see below) this indicates that the space of 'near functions', or similar functions with small discrepencies in bit values, is even smaller that the space of exact functions.

4.0.3 An open Question

It remains an open question as to whether there exists some measure of a P_1 or P_2 that can be used to determine its value in the search space. It may be possible that with an arbitrary postconditions(P_1) we could somehow recognize whether this fragment could begin F in a program of less than N instructions. Perhaps it is possible to genetically produce such an algorithm for a given instruction model. However, the fundamental operations on F and postconditions(P_1) would be of a counting and Boolean algebraic nature. These very general operations, utilized on very general Boolean equations, seem unlikely to result in an algorithm capable of determining the fitness of a P_1.

5 Functional Unit Based Search Space Pruning

The second goal of the project was to complete a machine-independent superoptimizer. The simulation model of superoptimization attempts to be machine independent while maintaining efficient speed. In addition to being machine independent, simulation has the following advantage over direct execution:

> If we store the data state between the simulation of each instruction, then for a program of length L, if we change only the Lth instruction of the program, we need only re-execute the new Lth instruction to determine the input

This fact allows us to make some speed gains against the set-back that we must calculate output bit-by bit, rather than in the bit parallel of actual execution.

Our resulting code generates 2,500 programs per second (for programs of length 4.) When factoring for computer speed differences, this makes our code a factor of 100 slower than Massalin's machine dependent superoptimizer[**?**] which generated 50,000 program per second. (Within automatic inlining our code is a less than 100 times slower, though measurements have not yet been taken.)

The resulting superoptimizer can generate short program sequences for any instruction set on any machine. The language form for bit level instruction semantics is presented in the infrastructure section of this document.

5.0.4 Functional Partition of Program Space

An interesting question raised about the space of short program fragments is, how functionally redundant is the space? In other words, do many program fragments code the same function, or do most program fragments represent (perhaps trivially) different functions?

Perhaps if the function space of programs is much smaller than the program space, we might be better off finding a semantic representation to search this space, that can map to a superoptimal code form. BTL is significantly expressive to represent this space, but can represent many functions not presentable by short programs do to its level of resolution. Functions representable by short programs are necessarily constrained to a higher degree of bit-parallelism than BTL assignment lists.

Figure 1 is a graph of the size of the program space, as graphed against the length of the program space, for the m68k. The dashed line represents the number of unique functions that make up this program space. In this graph, we are searching the program space of functions with a single input variable. As can be seen from figure 2, the redundancy (programs over functions) with program length is increasing on the m68k. While we would like to extend this data set further, this analysis is expensive and becomes impracticle for larger program lengths.

This result suggests that searching the instruction space for superoptimal functions is a highly inefficient approach. It would be a much better idea to search the function space itself. Figure 3 presents the distribution of redundancy for length 4 m68k programs of a single input variable. The number of programs determining a function is graphed on the x axis. The number of functions with this redundancy is given on the y axis.

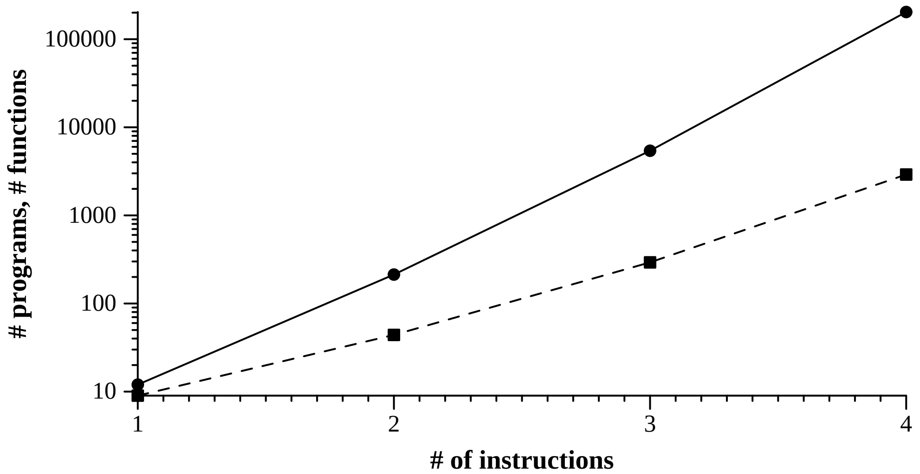

Figure 1: 1 input variable Program and Function (dashed) space size with program length in m68k machine instructions

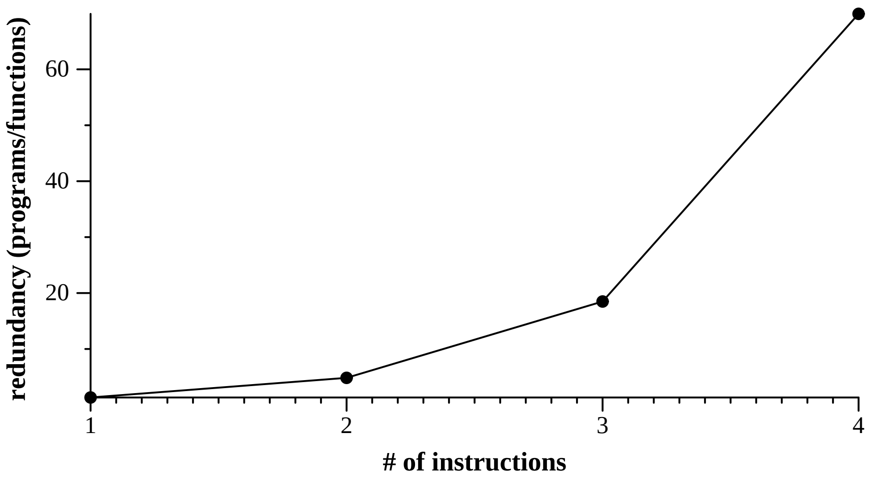

Figure 2: 1 input variable Function Redundancy (programs/functions) with program length in m68k machine instructions

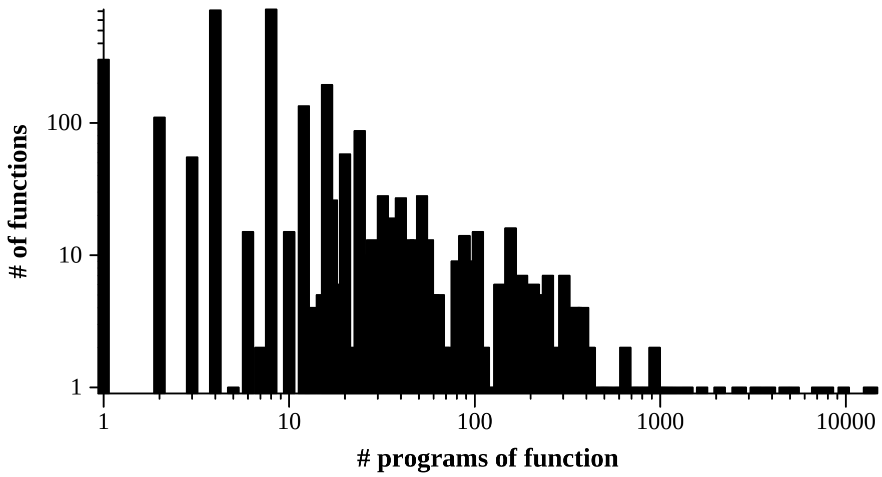

Figure 3: 1 input m68k function redundancy distribution for 1 input programs length 1-4 in m68k ALU instructions

Data for 2 input functions suggests a similar trend in program and function count with program length- as shown in figure 4. Figure 5 shows the redundancy rate with program length for the 2 input case. Figure 6 shows a breakdown of program redundancy in the number of functions.

This indication of the size of the function space provides for an interesting method to adjust the search space size. A program such as *signum()* could be generated with only 2 functional units of length instructions maximum, where the search space size factor is 300.

Recall that for individual ALU instructions on the m68k, we determined a branching factor for programs of 30. We can show that

$$30^4 \simeq 300^2 \tag{13}$$

For longer optimal programs, the search space size for program space becomes larger than the search space for functions.

Figure 7 compares the number of functions with program length for the one and two input cases. Figure 8 shows the redundancy of 1 and 2 input function spaces with program length. As can be seen, with increasing inputs the rate of redundancy does not increase as quickly with function length.

This experiment is repeated for 1 and 2 input on the alpha asa presented in graphs 9 through 12.

Based on these results we present the following model for superoptimization utilizing representable functional units rather than machine instructions:

1. **compile-compile time:** Generate and store all functions from the set of all programs up to length N.

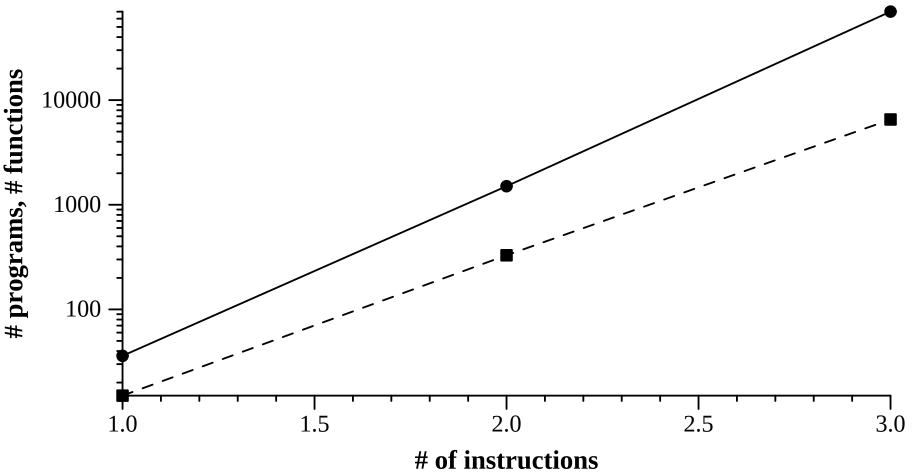

Figure 4: 2 input m68k number of programs and functions in search space of up to x program length

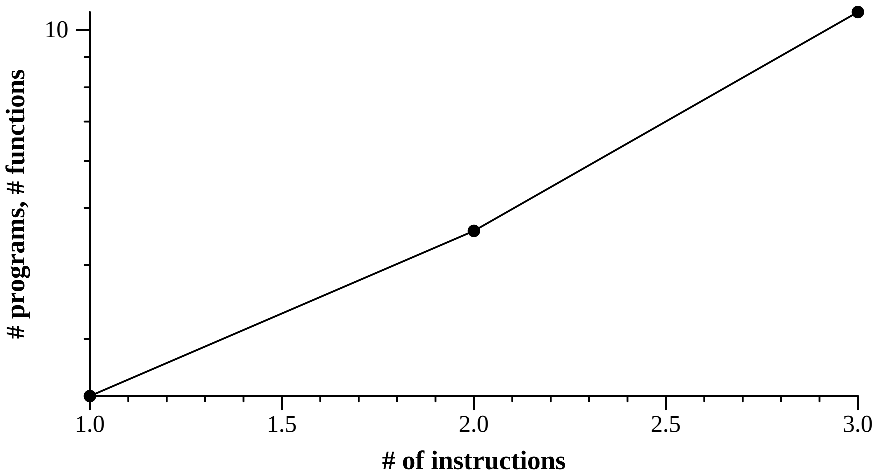

Figure 5: 2 input m68k function redundancy distribution

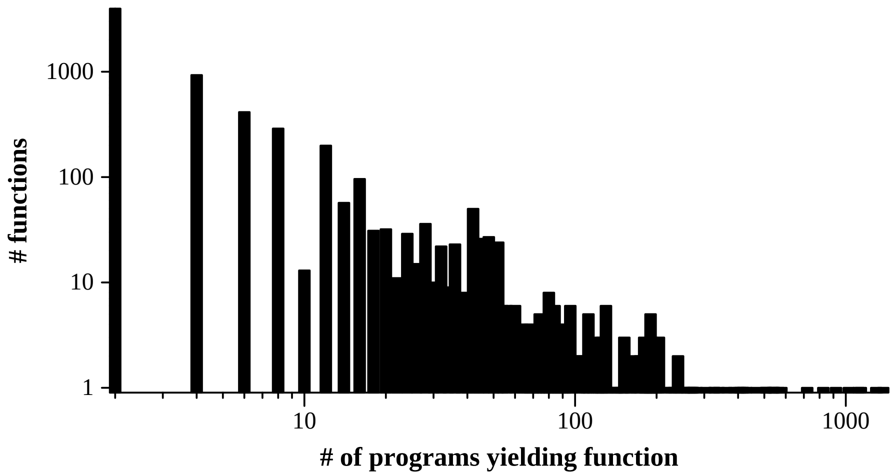

Figure 6: 2 input m68k function histogram of functions haveing number of programs

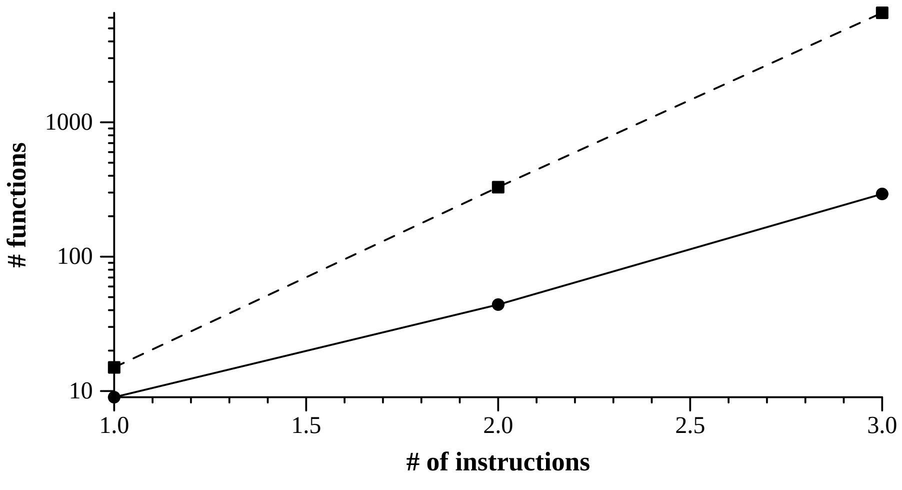

Figure 7: comparison of 1 and 2 inputs (DASHED) m68k function space size with program length

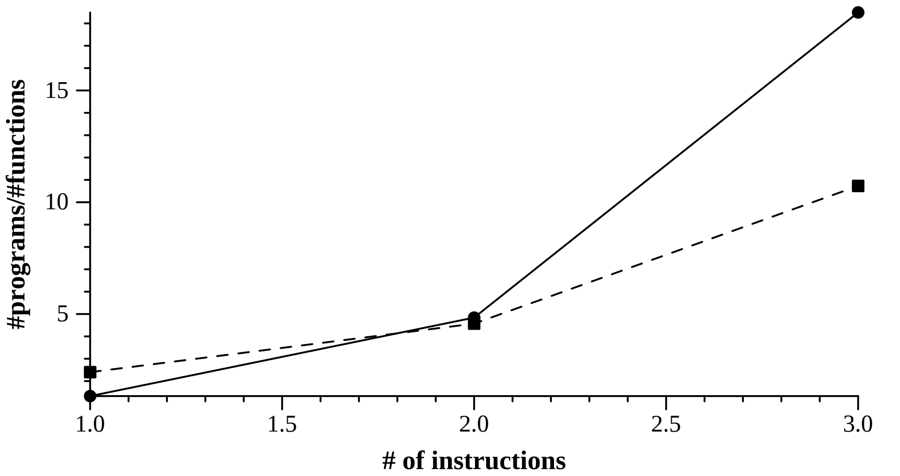

Figure 8: comparison of 1 and 2 inputs (dashed) m68k function space redundancy with program length

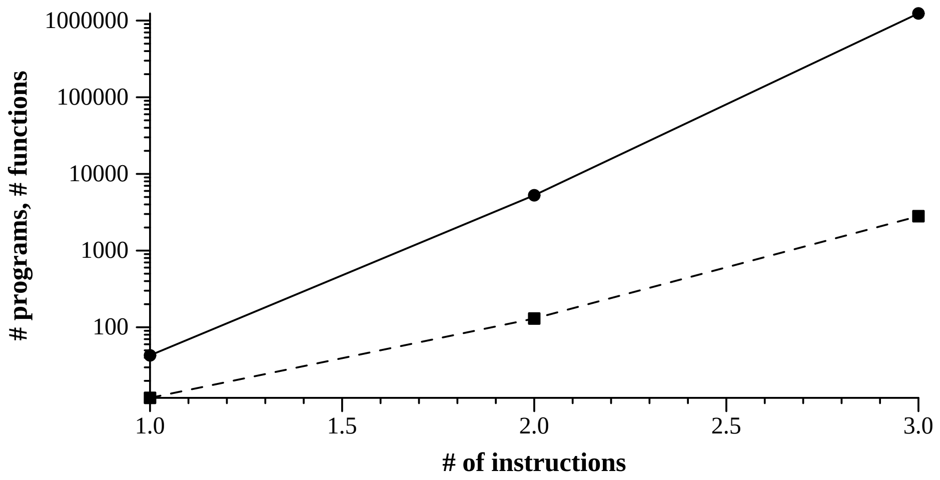

Figure 9: ALPHA function space and program space size for 1 input with program length

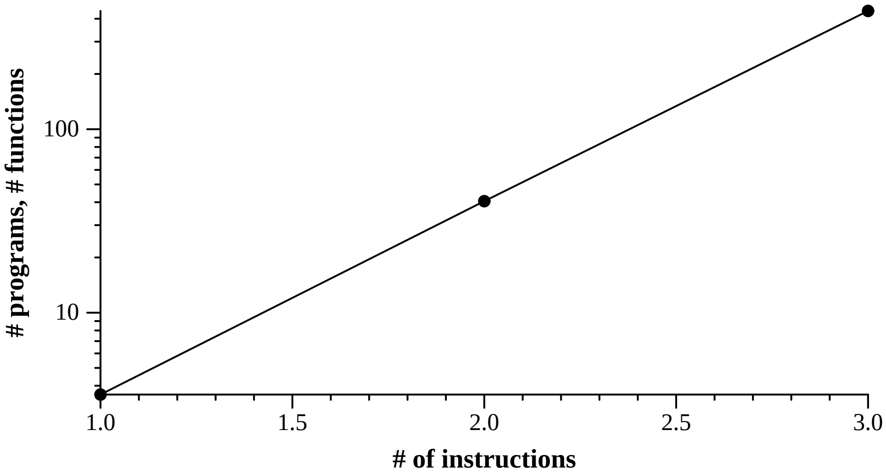

Figure 10: ALPHA function space redundancy with program length

- Store the input and output locations of the function, and there read/write properties (just as we did for machine instructions in our SAL language (see below)).
- Store the minimal cost program form found along with the function (for use in code generation).
- Order the functions so that functions by cost (length) of program representation.

2. **compile time** Superoptimize for program fragment over exhaustive search of up to M function units.
   - Load the function set
   - Do a breadth first search of all functional combinations (as with standard superoptimization on machine instructions.)

Figures 13 and 14 show search space size vs. program length calculated in a simple simulation, modelling program space size by the equations presented earlier. In each case, functions are generated for programs of up to length 4.

As usual for short program sequences we are at an advantage with a smaller functional space to search. However, this advantage is lessened for smaller program lengths making up our functional partition. Figure 15 represents the simulation of search space sizes when we utilize the $6,500$ functions of programs with 2 inputs up to length 3, rather than the $50,000$ of up to length 4.

This experiment is repeated in figures 16 and 17 for the ALPHA ALU instructions, excluding multiplication and division.

The following is a schematic of the degree of overlap of the MIPS and ALPHA instructions utilized:

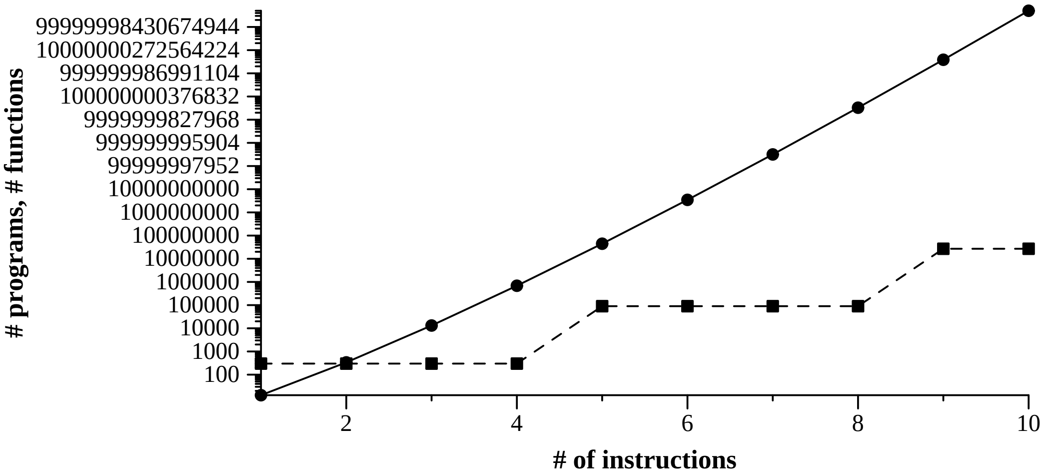

Figure 11: a simulation of search space size for program and function components for 1 input on the m68k.

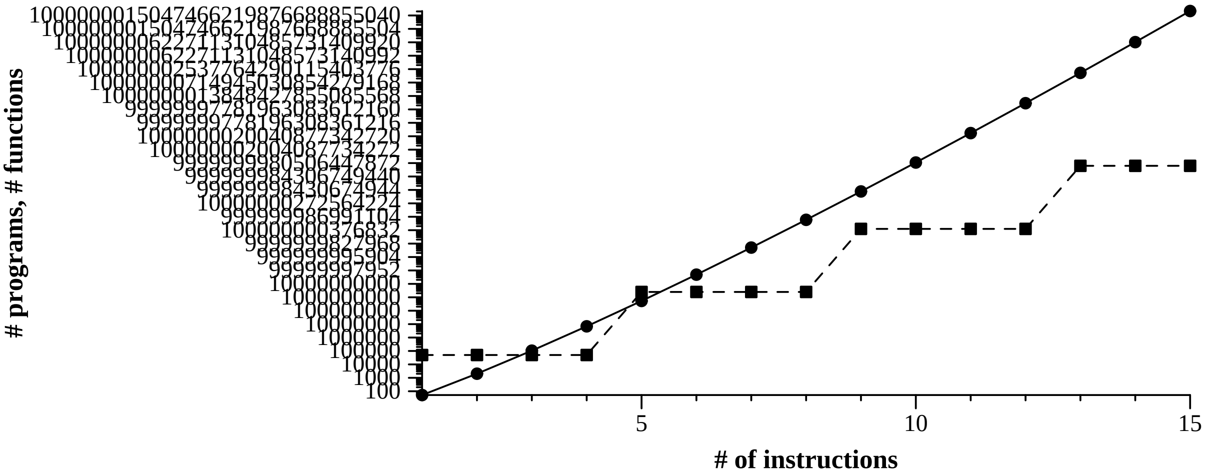

Figure 12: a simulation of search space size for program and function components for 2 inputs on the m68k.

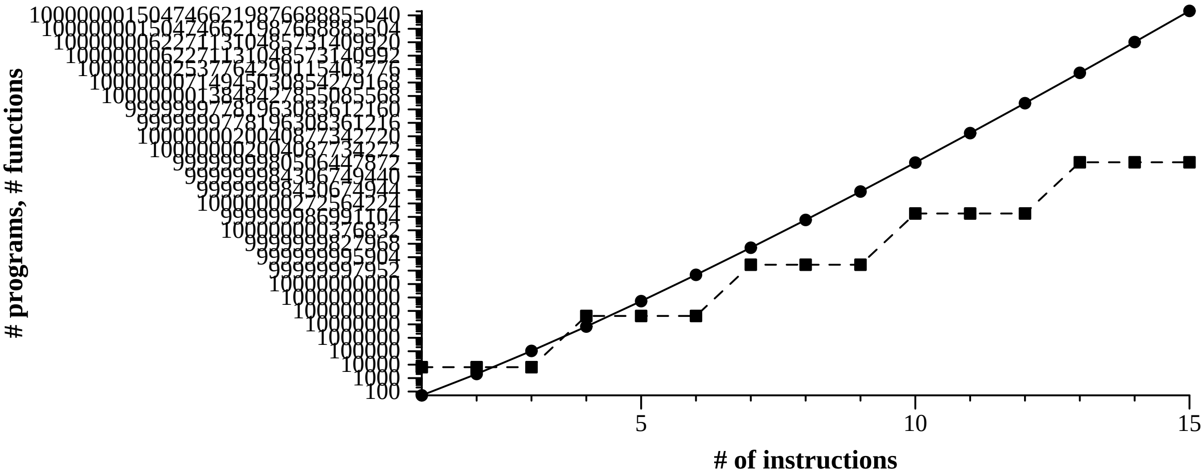

Figure 13: a simulation of search space size for program and function components for 2 inputs with functions up to 3 instructions with functions up 2 intstructions on the m68k.

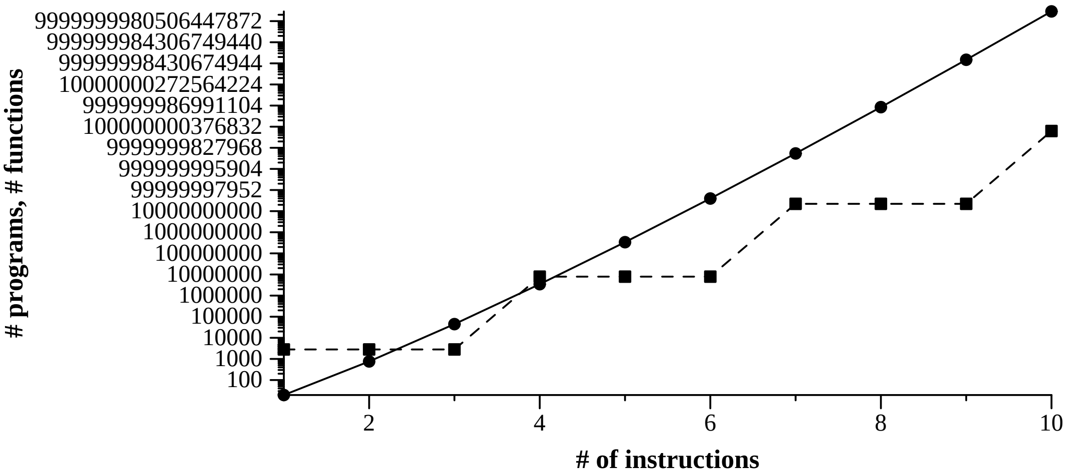

Figure 14: a simulation of search space size for program and function components for 1 input on the m68k.

```
M68K    ALPHA
----    -----
SUB      SUB
ADD      ADD
SUBX
ADDX
OR       OR
AND      AND
EOR      XOR
         ORNOT
         ANDNOT
         EQV
NOT      NOTPSEUDO
MV       MVPSEUDO
TST
NEG
NEGX
CMP      CMPLT
         CMPLE
         CMPEQ
```

Figures 18 and 19 compare the space sizes for the m68k and ALPHA for the 1 input case and 2 input case, respectively.

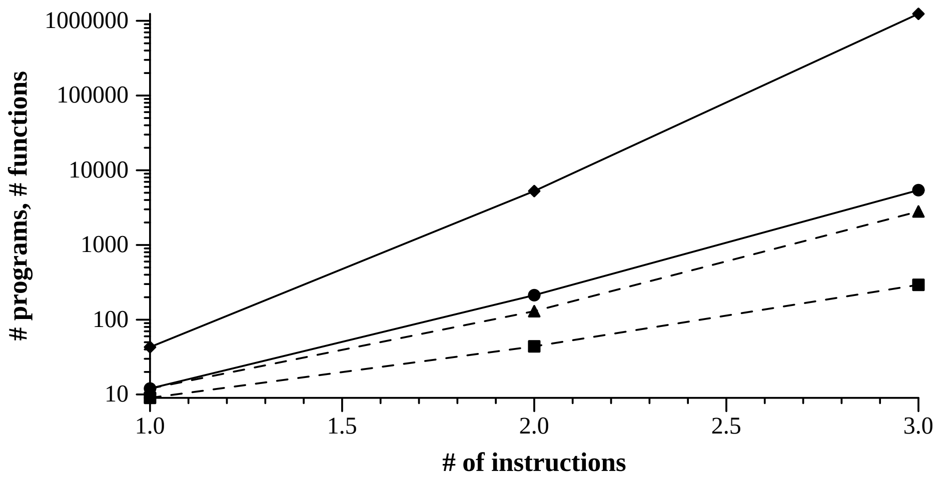

Figure 15: a simulation of search space size for program and function components for 1 input on the m68k.

Some calculation must be done to determine the cost of generation of a function set. Larger function sets over slightly larger program-fragment sizes do not seem to have much advantage in this simple simulation. It may be that an advantage for 2 inputs only arises for functions generated for program fragments up to length 5 or 6. In particular, the rate of redundancy in the function set seems to determine the rate of divergence of the program and function search space size. At legnth 5 or 6, 2 input functions should be come highly redundant (noted by an greater than linear climb in the redundancy graph, as seen in figure 2) How large a function set we can effectively manipulate in memory is a major concern for this technique, as well as the time for a large search set regardless of growth rate.

6 Future Work

6.1 Superoptimization By Searchign Function Space

The function space is asymptotically smaller than the program space as a function program length, and the rate of growth is less as well. It was shown that the function-space representation can be represented explicitly by partitioning the program space of a simulated machine instruction set for small program fragments to some length n. The result allows us to utilize an existing machine instruction description, such as $\lambda - RTL$[?] to synthesize the function partition. It further automates the process of exact function discovery, removing program space redundancy for local program segments up to length n. From the machine description-generated function description, we can apply superoptimization on a more efficient search space.

6.1.1 A Machine Simulator

A simple λ-RTL simulator will supply the mechanism for paritioning of the instruction space into a functional space. A program to exposit all possible instruction forms from the λ-RTL description will generate the initial program-instruction search space. Run on the simulator, we can then partition the redundany program space into a functional unit space.

6.1.2 Searchspace Unit Compilation

To further speed up superoptimization, the functional units can be 'compiled' for the native code of the machine on which superoptimization is executed. This will execute functional units with a speed closer to that of machine-instruction driven superoptimization.

6.2 Semantic Equivalence of Program Fragments

Semantic equivalence is determined for two program fragments P_1 and P_2 when there postconditions are shown to be functionally equivalent Boolean statements. An infrastructure for this is fully developed, such that with BTL model, the equivalence operation between the two code fragments can be easily determined. Symbolic evaluation, post condition reduction, supports any code fragment of small length (basic block length) in effective time. Loops can not be handled by this technique. Outside the basic block, forward conditional branching, is easily handled by the current implementation, while backward (looping) branches are not. The simulation model can be used to determine equivalence on looping code.

6.3 Optimizer Debugging

With this capability, we can check and guaruauntee that transformed basic blocks are equivalent to their original form. For example, we could easily apply this technique for compiler-infrastructure development as a debugging strategy. In VPO, the very portable optimizer, we could apply semantic equivalence verification on each transform, highlighting discrepancies in the RTL code as it is applied. VPO has a viewing tool to watch transformations take place. SAL could be applied to guaruauntee semantic equivalence in this tool.

6.4 Dynamic Instruction Set Computing Optimization

Recently, reconfigurable hardware has become of interest. Utilizing post condition generation, Boolean algebra to represent any short function can be created. On a machine architecture that could accept such fragments as special-programmable instructions, kernel code of programs could be superoptimized to the Boolean algebraic level.

6.5 Boolean Reduction With More Expressiveness

There are unique opportunities to 'humanize' computation in Boolean logic if a way can be implemented to reduce Boolean equations to minimal form while maintaining the bit-level parallelism constructs of arbitrary BTL assignment statements. Such a mechanism may not exist, or may be too complex too implement to practical effect.

7 Conclusions

Bit level semantics appears to provide no opportunity for pruning the superoptimization search space. In contrast to a lower level semantic notation, it appears a more appropriate representation for a symbolic superoptimizer would be a symbolic machine description language such as λ-RTL. This language can be used to synthesize, for a given machine, a partition of the short instruction sequences into functions. These functions make an ideal construction unit for a smaller superoptimization space. This method is a superset of the adhoc methodology for legal instruction tables generated by Massalin. Since this search space reduction can be performed at compile-compile time, with compiled function-representations as output, it should lead to significant improvement in symbolic superoptimization performance. Furthermore, it is safe from the inadvertant elimination of useful code sequences by and ad-hoc table driven approach. Future work in superoptimization can take advantage of this model of search space construction.

Overall we have developed a toolkit for studying the bit-level properties of a machine-instruction program space. Superoptimization and instruction-search space analyses can be run simply by loading in an SAL description of a machine instruction set, a program fragment to generate superoptimal code for (or other similar input), and the creation of a Sosearcher object to perform the search/analysis. Generation of various kinds of SOSearch objects allow us to generate many different superoptimization analysis programs and techniques with less code-rewriting. We have further developed a more advantageous model of symbolic superoptimization that focuses on the functional level of program fragments.

8 Appendix A: Infrastructure

The infrastructure for our Symbolic Superoptimizer is divided into several layers of abstraction. At its lowest level, it contains an effective language for the computation and reduction of Boolean expressions. This expression model is well suited for register-style storage locations of Boolean values (bits). The second level provides a model for the expression of bit-level semantics of an instruction set, and program fragment execution and symbolic evaluation based on this instructions set. The third layer presents various methods of superoptimization, the models of which were discussed in this document. Future work in superoptimization could easily extend from the existing infrastructure.

8.1 Bit Transfer Language: BTL Module

Bit transfer language, or BTL, as we shall call it, is designed to represent the bit-level semantics of machine architectures and low-level intermediate representations in the form of guarded Boolean assignment statements. In a sense, BTL is a parallel to RTL in syntax. Semantically, it is quite different, in that there is not necessarily a one-to-one function between an BTL expression on the RHS of an assignment, and an instruction, since we are working at the sub-instruction semantic level.

8.1.1 Storage Locations

The most basic component of a BTL statement is the storage location. Storage locations are a declared name and length in bits. Global storage locations are also given a type for parameter matching at the SAL level. Thus, storage locations, such as registers and flags might be declared as

```
DEFINE conditionFlag N[1], Z[1], V[1]
DEFINE generalRegister D0[Q], D1[Q], D2[Q]
```

where Q is the length of registers in number of bits. Once in a symbol table, these symbols can be used in Boolean expressions.

When a storage location appears in a BTL expression, the quantity in the bracket (the array expression) represents the bit position of the storage location. Thus

```
D0[i]
```

refers to the *ith* bit of *D0*. The expression within the index of a symbol location can be any simple linear algebraic statement utilizing index variables, constants, and the plus and minus, multiply and divide operators. Thus

```
D0[i+j-1/k-4*m]
```

is a legitimate bit access within the scopes of the index variables i, j, k, and m. Index variables are described below.

8.1.2 Boolean Expressions

The core of BTL are Boolean expressions. In our representation we support 'AND', 'OR', and 'NOT', operations. BTL is recursive in structure, representing everything up to guarded assignments as an abstract syntax tree. The OR operation is unary in that it takes one child node expression as an argument. 'AND' and 'OR' are m-ary, in that they take an arbitrary number of child-nodes.

The arguments of Boolean expressions may either be storage location, or numeric constants.

BTL supports prefix syntax for the unary negation operator. It support a prefix and infix notation for conjunction and disjunction. Thus, the following is a legal BTL expression:

```
{ + | B[1]+B[2]*256* (C[1]+C[2]), A[2], A[3]+1024}, {* | B[2]*B[3], C[4]}
```

where parenthesis override default infix operator precedence. This flexible notation allows succinct representation of machine instructions, as we shall see when we introduce indexed expressions below.

8.1.3 Index Expressions

The prefix form of the AND and OR operators support bit-level sequence indexing. This is done with bound index variables bound to the AND and OR operators of a BTL expression.

Thus

```
A[1]+A[2]+A[3]+A[4]
```

has the equivalent form in BTL of

```
{ + | j=1..4 : A[i] }
```

while the expression

```
A[1]*B[1]*B[2]*B[3] + A[2]*B[1]*B[2]*B[3] + A[3]*B[1]*B[2]*B[3]
```

has the equivalent form

```
{+ | i=1..3 A[i]*{ * | j=1..3 : B[j] } }
```

The declared index ranges of bound index variables must be numeric sequences. Other index variables cannot be placed in these sequences. Overlapping lists and ranges are allowed, in combination, as in the following example

```
{ + | inc=0..4, 1, 2, 9..12, 15 : A[inc+2] }
```

Index variables are bound variables whose scope is bound by the operator to which they are attached. There names must be unique within there respective scopes upon declaration (after this it is irrelevant, although unique names are important for comprehensible human-form output.)

8.1.4 Assignments

A BTL statement is a guarded assignment. Each BTL statement has 3 arguments. The first is a BTL expression representing the guard of an assignment. The assignment is performed if the guard evaluates to true, otherwise it is not performed. The second argument is a symbol location instance, indicating the bit position that will be stored to (in other words a LHS argument.) The third argument is a right hand side- a BTL expression that is evaluated; the result of which is stored to the LHS argument.

The assignment operator may also have a bound index variable, which is applicable to all the arguments of the statement.

The following is an example BTL assignment:

```
D[i+1] = (i=0..3, 7) A[i+2]+B[i+1]+ {* | j=0..3 : C[i+j] }
```

Assignment is only features in an infix syntax.

8.2 Preconditions and Postconditions

A precondition/postcondition is an Boolean expression that tests for equivalence. It takes two arguments, both of which are Boolean expressions. It evaluates to true if its two argument expressions are always equivalent, otherwise it evaluates to false.

A BTL precondition/postcondition can have a bound index variable, just like a BTL assignment.

8.3 Lists

BTL supports lists of assignments, preconditions and postconditions. These lists are separate in that a list may consist of only assignments, or only preconditions/postconditions.

A BTL assignment list can be forward substituted into a list of postconditions. Postconditions can be backwards substituted on each other to generate weakest preconditions.

8.3.1 Order Stamping

BTL assumes assignments are performed in the same order in which they are listed. At each assignment, the storage state of the location is maintained with an 'order stamp'. Order stamps represent intermediate states of storage of locations. This is important for full forward propagation, for the generation of symbolic post-conditions. Order stamps are represented by proceeding a location reference with a '_NUMBER' tag, in which 'number' represents the intermediate state index of the location.

Thus the following set of RTLs as stated by the user

```
D0[i] = (i=0..3) A[i]+B[i]
D1[i] = (i=0..3) A[i]*D0[i]
A[1] = 0
D1[i] = (i=0..3) D1[i] + A[i]
```

will translate to a form with order stamps as

```
D0_1[i] = (i=0..3) A_0[i]+B_0[i]
D1_1[i] = (i=0..3) A_0[i]*D0_1[i]
A_1[1] = 0
D1_2[i] = (i=0..3) D1_1[i] + A_1[i]
```

Note that in this case, the unspecified bit values of $A1_1$ are assumed to propagate from the values of $A1_0$.

8.3.2 Assignment Semantics and Recursive Assignment

Across operations with index variables, evaluation is always considered to be done in parallel for all subexpressions. In other words, all 'bits' indexed in an expression by index variable incrementing are assumed to be evaluated simultaneously. The same applies for assignment statements.

In some situations we may need to represent assignment from 1 bit of a register that is being assigned to another bit of the same register. As we discussed above, automatic intermediate order stamp generation would take the following assignment:

```
CC[1] = (i=1..3) A[i]*B[i] + A[i]*CC[i-1] +B[i]*CC[i]
```

,and spit out the following:

```
CC_1[1] = (i=1..3) A_0[i]*B_0[i] + A_0[i]*CC_0[i-1] +B_0[i]*CC_0[i]
```

,where what we would like is:

```
CC_1[1] = (i=1..3) A_0[i]*B_0[i] + A_0[i]*CC_1[i-1] +B_0[i]*CC_1[i]
```

In order to produce the desired intermediate evaluation, we must specify intra-order bit recursion explicitly. We utilize '&' as a prefix operator to denote 'intra-order' recursion between bit positions of a location. Thus we would write the above statement as

```
CC[1] = (i=1..3) A[i]*B[i] + A[i]*&CC[i-1] +B[i]*&CC[i]
```

to produce the desired order stamping.

8.3.3 Expressiveness

This concludes the syntax of BTL. It allows us to elaborate the bit level semantics of a machines instruction set as lists of boolean assignments. Program fragments can generate postconditions and preconditions with symbolic forward and backward, respective, propagation of symbolic data values.

What kinds of things can't be represented?

8.3.4 Forward Substitution

Given a list of guarded assignments, forward substitution is the process of eliminating intermediate assignments from the right hand side of each assignment. After forward substitution, a functionally equivalent list is presented which states each assignment as an expression on only input values.

Forward substitution is accomplished through recursive forward propagation of data values. It is complicated in our scheme by the presence of bit-index expressions. In general forward substitution results in break-down of assignments over multiple bit-indexes to single-value bit indexes.

8.3.5 Backward Substitution

Backward substitution is the process of determining preconditions necessary for an assignment list to satisfy some known postconditions. This presents a mechanism for the classic Hoare precondition analysis technique for program verification.

8.3.6 Evaluation

Evaluation takes a data-space of input values, and computes a data-space of output values from a given assignment list.

For a BTL expression, evaluation takes the current index variable and data state and computes the Boolean value of an expression. This includes precondition/postcondition expressions. Assignment statements are not expressions and cannot be evaluated.

8.3.7 Execution

Assignment statements are executed, by taking an input data space, and evaluating all subexpressions to compute an output data space. This execution includes the evaluation of each assignment statement and expression over all of any index variable values it may bind.

8.3.8 DNF form

BTL currently supports reduction to DNF form. This reduction may further reduce index variable/bit-parallel notation of an initial BTL assignment statement.

8.3.9 Data Storage

The BTL module supports a data space concept. A data space inputs a symbol table and provides storage for each symbol, including separate storage for each order of each symbol(for symbolic analysis.)

8.4 Reduction: QM Module

BTL supports the ability to reduce arbitrarily complex BTL assignments to minimal DNF form. This is an important feature for effective representation of postconditions. However, there is currently no way to reduce such expressions while maintaining the bit-level parallel notation (index variables). Thus reduced DNF BTL assignments take the form of explicitly stated minterms with simple constant bit index expressions.

The code to support reduction uses the Quinne-McCluskey method, and can be found in the 'qm' module.

An area of exploration was to attempt to maintain maximum bit parallelization of a symbolic representation. Use of a QM algorithm resulted in a halt of exploration in this area.

8.5 Semantic (Symbolic) Assembly Language: SAL Module

The next level of our hierarchy allows the description of ALU and control flow like instruction sequences in terms of a list of BTL boolean assignments.

The SAL level can be divided into three relatively separate components. The representation of a machines instructions set, and the representation of program fragments for either symbolic or simulation analysis.

8.5.1 Instruction Set Representation

Typed Storage Locations When a storage location is declared, it is declared with a type. Similar registers should be declared with the same type. For example

```
DEFINE generalReg D0[4], D1[4], D2[4]
DEFINE generalreg D3[4]
```

declares a general register type with several members. Subtypes can be declared by the declaration of a type that consists of other types. For example

```
DEFINE regt TYPE generalReg, TYPE floatReg, SP[4]
```

declares a type reg which includes general registers, floating point registers and a 4 bit location called SP.

Typing of storage locations is important in machine instruction description in order to describe legal parameters for an instruction form.

Instruction Format Declaration Instruction forms are declared with the superoptimizer as a opcode name, a series of formal parameters, a declaration of local temporary storage locations (those that do not persist between instructions), a list of BTL (guarded) assignments that perform the instruction, and a list of guards that are activated until a label is reached on some future instruction instance (used to implement control flow instructions.)

As examples of instructions we present two from the M68k instruction set:

```
INSTRUCTION SUBX genReg A[Q], genReg B[Q] =
  LOCALS CC[Q]
  CC[0]=~X
  {i=1.. Q-1 : CC[i]=~A[i-1]*B[i-1] + ~A[i-1]*&CC[i-1] + B[i-1]*&CC[i-1] }
  C = ~(~A[Q-1]*B[Q-1] + ~A[Q-1]*CC[Q-1] + B[Q-1]*CC[Q-1])
  V= ~B[Q-1]*A[Q-1]*CC[Q-1] + B[Q-1]*~A[Q-1]*~CC[Q-1]
  X=C
  {
   i=0.. Q-1 :
              B[i]=~A[i]*~B[i]*~CC[i]+A[i]*B[i]*~CC[i]+A[i]*~B[i]*CC[i]
                   +~A[i]*B[i]*CC[i]
  }
  N= B[Q-1]
  { * | i=0.. Q-1 : ~B[i] } -> Z=1

INSTRUCTION BGT LABEL JumpPoint =
  BLOCK UNTIL JumpPoint IF ((N*V)+(~N*~V))*~Z
```

In the first instruction, 'SUBX', its two operands can be any general purpose m68k register. (the immediate field form of the instruction is not implemented here.) A local variable CC, representing the storage of cascading carry bits, is declared in the instruction. The subtract operation of parameter A and X from B is then represented as a cascading subtract. (The semantics are the same regardless of actual implementation method.) The instruction description includes description of the setting of condition bits such as a zero flag (Z) and a negative flag (N).

The second instruction presents the 'BGT', branch if greater than, instruction. This is represented symbolically by a guard expression, which blocks execution until a label, referred to by the label formal parameter 'JumpPoint', is encountered in the program fragment.

With this instruction format, all ALU and most branching instructions can be represented.

8.5.2 Symbolic Program Fragment Representation

In a symbolic program fragment, a set of instruction opcodes and actual parameters is given in a sequence. The data structures automatically produce a set of minimal post-conditions for this sequence, representing the semantic action, at the bit level, of the program fragment.

8.5.3 Simulated Program Fragment Representation

A simulated program fragment is also a list of instruction opcodes and there actual parameters. However, post conditions are not generated. Instead, a data-storage space is maintained at each instruction instance, to represent the simulated data result at the end of the given instruction simulation.

8.5.4 Modularity

A program fragment can be generated symbolically or for simulation simply by creating a SALProgram object of the appropriate type, and adding instruction instances to the program objet.

Simulation is a simple function call, while classes exist to provide simple incrementing or importing of specific data values, over the input data-set for the program.

Functions are also presented to compare the symbolic or simulated output of programs.

8.5.5 Output Comparison

SAL can compare the output of two functions and determine if they are equivalent. This comparison is performed on perumtations of input and output mapping, so that the comparison is not considered with the partitcular cell locations of input and output, so long as they map consistently.

8.5.6 Output Correlation

SAL contains functionality to correlate the output of two or more simulated SAL program fragments. Correlation is returned as a percentage, indicating the absolute value degree of output correlation between the functions represented by the two code fragments. Correlation is invariant over rotation of data in output storage spaces, and is taken for the maximum correlation found in a consistent mapping of each functions inputs and outputs. Absolute correlation considers the compliment of a value to be completely correlated with the value. This form of correlation is the most useful in detecting programs containing similar output information content.

8.5.7 Function Sorting

SAL can maintain partition a group of SAL simulated program fragments by their semantics (input-output function.) This can be utilized to examine the distribution of functions over a program space. Function sorting utilizes a function-space data structure to create the partition.

8.6 Superoptimizer Layer: SO Modules

The Superoptimizer level comes in several flavors. The four available currently perform symbolic and simualtion based search-space traversal. Of pure superoptimization, implemented are 'symso', which performs pre-condition based symbolic superoptimization, and 'simso' which performs instructions simulation. 'corrso' implementes our correlation experiment outlined earlier in this document. 'sizeso' performs the program partitioning discussed earlier in this document.

All versions rely on a class called SOSearcher, to search the space of programs to a desired length. The versions of code differ only in the generation of legal instruction in the search space, the order of search space traversal, and the technique of program fragment analysis. All searches are in a breadth first manner with respect to program length n.

8.6.1 symso

'symso' generates all possible programs up to length m to test for an equivalent program. Programs are generated from the last instruction back towards the 'beginning' of a correct program fragment. Preconditions are generated as each level of the program length is generated.

This version of the superoptimizer generates all program options for level 1 (the last instruction position.) It then chooses a branch of the space to explore, and generates all options for that branch at level 2 (the second to last instruction position.) It then chooses a branch of this to explore and generates all instruction options at level 3, and so on to level n (the first instruction of the program.) Assuming failure occurs at level n, backtracking down to level 1 is applied. Thus we would backtrack to try the next instruction at level n, and then failing for all possibilities here, go back to try the next instruction possibility at level n-1. This requires some storage space, but not the massive space requirements of full search-space storage. It eliminates redundant generation of preconditions at each level of the search. Instruction options are only pursued in the case that their preconditions are legal (which as was explained in the second section, is almost always the case starting at level 2.) Upon detection of a correct program ,we are guaruanteed that no shorter sequence of the modeled instructions exists which would perform the desired function.

8.6.2 simso

'simso' operates by searching the search space in a forward direction. Its levels are in direct (not inverted) correspondence with instruction order in the test program fragment. Level 1 represents the first instruction options, level 2 the second, and so on. The search takes place with the same backtracking as discussed above. The resulting search is depth first on all programs of length m. We search programs from length 1 to n in breadth first. This is identical to the operation of sosym above, except with regard to forward program generation rather than backward generation.

A simulated test program need only update the simulation for the last instruction changed, assuming only one input was required to run to compare the test program with the reference. This is usually the case [?]. Thus we are generally simulating only one instruction per program searched.

8.6.3 corrso

'corrso' replaces comparison of outputs of a program with the reference with correlation. The maximum correlation of a mapping of a program fragment's output locations to the reference function outputs is calculated. Further, this maximal correlation is made invariant over output bit rotation or storage-location wide negation. The result is a correlation which measures the information content-agreement between the given test fragment and the reference fragment.

The only difference between this code and simulation code is that that SAL comparison routine in the main search loop is replaced by a call to the SAL correlation routine.

8.6.4 sizeso

'sizeso' takes as input the usual instruction description, and the number of inputs that each function will have in program generation. (A specified number of registers assumed to contain data on input as we generate a program.) From this, it generates the search space to length n, placing all programs into a functional partition. Each program is catagorized by its functional semantics, and the quantity of programs representing each function is recorded.

Statistics are then generated by 'sizeso' to describe the functional distribution of the search space programs. Each program is stored in its functional partition, so that each functional unit can be examined for its program fragment content.

9 APPENDIX B: Syntax

The following sections provide BNF grammars for BTL and SAL input languages. The BTL language is just a subset of the input language for SAL. The grammars given here are extracted from the yacc grammars for the input language parsers.

The grammar for order stamp encoded SO-level input language is not given in this document, as it is strictly an internal representation (not written directly by the end user.)

9.1 BTL

9.1.1 Symbol Instances

Symbols appearing in RHS expressions and the LHS of assignments are allowed as follows:

symbolOrder $\longrightarrow$ _ **number number**| λ
bitLocation $\longrightarrow$ **locationName** | **ident** [**indexExpression**]
indexExpression $\longrightarrow$ **number** | **indexVariable** |
  indexExpression + **indexExpression** |
  indexExpression $-$ **indexExpression** |
  indexExpression $*$ **indexExpression** |
  indexExpression / **indexExpression** |
  (**indexExpression**)
LHSsymbol $\longrightarrow$ **bitLocation symbolOrder**
RHSsymbol $\longrightarrow$ **bitLocation symbolOrder** | & **bitLocation symbolOrder**

where location names are defined in a BTL symbol table.

9.1.2 Boolean expressions

Boolean expressions are defined in a usual way to create the expected operator precedence on infix operations.

expression $\longrightarrow$ **level3Term**
level3Term $\longrightarrow$ **level3Term** + **level3Term** |
  prefixDisjunct | **level2Term**
level2Term $\longrightarrow$ **level3Term** $*$ **level3Term** |
  prefixConjunct | **level1Term**
level1Term $\longrightarrow$ **RHSsymbol** | **NUMBER**|
  level1Term | (**level3Term**)
prefixDisjunct $\longrightarrow$ { + '|' **indexVarDeclare expressionList** }
prefixConjunct $\longrightarrow$ { $*$ '|' **indexVarDeclare expressionList** }
indexVarDeclare $\longrightarrow$ **nonLocationName** = **indexList**
indexList $\longrightarrow$ **indexList** , **valRange** | **valRange**
valRange $\longrightarrow$ **number** .. **number** | **number**
expressionList $\longrightarrow$ **expressionList** , **expression** | **expression**

9.1.3 Assignments

Assignments (guarded and unguarded) are created with the following syntax.

assignmentList $\longrightarrow$ **assignmentList assignment** | **assignment**
assignment $\longrightarrow$ **unguarded** | **guarded**
unguarded $\longrightarrow$ **LHSsymbol** = **expression** |
  LHSsymbol = (**indexVarDeclare**) **expression**
guarded $\longrightarrow$ **expression** $->$ **LHSsymbol** = **expression** |
  expression $->$ **LHSsymbol** = (**indexVarDeclare**) **expression**

With the above grammar, all possible tree-form BTL assignment lists can be created.

9.2 SAL Instruction Set Description

An instruction set description consists of two parts, the storage location definitions, and the instruction set definition.

description $\longrightarrow$ **storageDeclarations InstructionDefinitions**

9.2.1 Storage Declarations

Storage location definitions are typed.

storageDeclarations $\longrightarrow$ **storageDeclarations storageDeclare** | **storageDeclare**
storageDeclare $\longrightarrow$ $DEFINE$ **unusedname declareList storageDeclare** |
$DEFINE$ **typename declareList storageDeclare** | λ
declareList $\longrightarrow$ **declareSymbol** , **declareList** | $TYPE$ **typename** , **declareList** |
declareSymbol | $TYPE$ **typeName**
declareSymbol $\longrightarrow$ **unusedname**[**number**]

Thus storage declarations of new types and extensions of types can consist of new storage locations and previously defined types.

9.2.2 Instruction Definitions

instructionDefinitions $\longrightarrow$ **instructionDefinitions instructionDefinition** |
instructionDefinition

SAL instruction set grammar allows the definition of instructions which accept only certain subsets of locations as actual parameters through a typed formal parameter syntax. Instructions can declare local storage locations, bit level assignments to evaluate, and guards to activate until a label is reached further in a program.

instructionDefinition $\longrightarrow$ $INSTRUCTION$ **unusedname formalParameters** =
localDeclares assignmentList labelGuardList
formalParameters $\longrightarrow$ **formalParameters** , **formalParameter** | **formalParameter** | λ
formalParameter $\longrightarrow$ **typename unusedname** [**number**] | $LABEL$**unusedname**
localDeclares $\longrightarrow$ $LOCALS$ **localList** | λ
localList $\longrightarrow$ **local** , **localList** | **local**
local $\longrightarrow$ **unusedname** [**number**] | **unusedname**
labelGuardList $\longrightarrow$ **guardActivation labelGuardList** | **guardActivation**
guardActivation $\longrightarrow$ $BLOCK\ UNTIL$ **labelparametername** IF **expression**

Locals are not typed as there is no reason to type them. Assignment Lists were discussed in the BTL grammar description.

This completes the instruction description syntax.

9.3 SAL Program Fragment Entry

To enter a reference program fragment one writes assembly language. The only difference is that instructions must be separated by semicolons, and a list of outputs must be specified.

program $\longrightarrow$ **programList outputsList**
programList $\longrightarrow$ **programInstruction** ; **programList** | **programInstruction**
programInstruction $\longrightarrow$ & **unusedname** : **instructionName actualsList** |
instructionName actualsList
actualsList $\longrightarrow$ **actualParamList** | λ
actualParamList $\longrightarrow$ **actualParamList** , **actualname** |**actualname**
actualname $\longrightarrow$ **unusedname** | **storagename** | **number**
outputsList $\longrightarrow$ **outputsList** , **storagename** | **storagename**

This concludes the discussion of the syntax of SAL and BTL input files.

References

[1] Davidson, J. & Fraser, C, Automatic Generation of Peephole Optimizations. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 111-116. ACM/SIGPLAN, June, 1984.

[2] Kessler, Peter Discovering Machine Specific Code Improvements. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 249-254. ACM/SIGPLAN, June, 1986.

[3] A Practical Method for Code Generation Based on Exhaustive Search. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 185-196. ACM/SIGPLAN, June, 1982.

[4] Massalin, Henry, Superoptimizer – A Look at the Smallest Program. In*ACM some conference*, pages 122-126