OPTGEN: A Generator for Local Optimizations

Sebastian Buchwald

Karlsruhe Institute of Technology
buchwald@kit.edu

Abstract. Every compiler comes with a set of local optimization rules, such as `x + 0` → `x` and `x & x` → `x`, that do not require any global analysis. These rules reflect the wisdom of the compiler developers about mathematical identities that hold for the operations of their intermediate representation. Unfortunately, these sets of hand-crafted rules guarantee neither correctness nor completeness. OPTGEN solves this problem by generating *all* local optimizations up to a given cost limit. Since OPTGEN verifies each rule using an SMT solver, it guarantees correctness and completeness of the generated rule set. Using OPTGEN, we tested the latest versions of GCC, ICC and LLVM and identified more than 50 missing local optimizations that involve only two operations.

Keywords: Intermediate Representations, Local Optimizations, Superoptimization

1 Introduction

Every compiler comes with a set of local optimization rules, like `x + 0` → `x` or simple constant folding. By definition, such rules exhibit a left-hand side of limited size and require no global analysis. Thus, they can be applied at any time during the compile run.

So far, the local optimizations provided by state-of-the-art compilers are incomplete. For instance, GCC 4.9 and ICC 15 do not support the local optimization `x | (x` $\oplus$ `y)` → `x | y`[1] whereas LLVM 3.5 fails to perform the optimization `-((x - y) + z)` → `y - (x + z)`. Furthermore, all three compilers miss some optimizations with non-trivial constants, like `x & (0x7FFFFFFF - x)` → `x & 0x80000000` for 32-bit integer types. Moreover, the compiler does not guarantee the correctness of the supported optimization rules. This raises the question for a generator that systematically enumerates and verifies all local optimizations up to a given pattern size.

On the assembly level, superoptimizers solve a related problem: They try to generate a better version for a fixed sequence of instructions, while preserving the semantics of the sequence. In order to guarantee the correctness of their transformation, they transform the instruction sequences into SAT or SMT formulas. Then, they use the corresponding solver to verify the equivalence of the constructed formulas. Later, generators for peephole optimizers used the same

[1] $\oplus$ stands for bitwise exclusive or, `|` for bitwise or, and `&` for bitwise and.

technique to verify correctness but also aim for completeness. However, they all used a limited set of constants, like $\{0, 1, -1\}$.

The dilemma of supporting all constants is revealed when creating constant folding rules. On a 32-bit architecture, we would create 2^{64} constant folding rules for every binary operation: `0 + 0` $\rightarrow$ `0`, `0 + 1` $\rightarrow$ `1`, and so on. Obviously, enumerating all these rules is far too expensive and impractical for end users. Thus, the handling of constants is the key challenge when aiming for completeness of the generated rule set.

In contrast to peephole optimizations, local optimizations work on the intermediate representation (IR). Since modern IRs in static single assignment form model data dependencies explicitly, local optimizations match these data dependencies instead of instruction sequences. This allows to perform local optimizations on patterns that span the whole function.

In this paper, we present Optgen[2], a generator for local optimization rules. Optgen takes a set of operations, their costs, and a cost limit as input parameters. It then generates *all* local optimizations up to the given cost limit and provides them in textual or graphical form. Furthermore, it generates a test suite that finds missing local optimizations in existing compilers. The contributions of this paper are:

- A generator for *all* local optimization rules up to a given cost limit.
- An approach how to cope with constants that can be backported to generators for peephole optimizers.
- An optimization that combines local optimization rules with global analyses.
- An evaluation of state-of-the-art compilers that reveals more than 50 missing local optimizations that involve only two operations.

The remainder of the paper is structured as follows. In Section 2, we discuss preliminaries and related work. Section 3 presents design and implementation techniques of Optgen. In Section 4, we combine local optimization rules with global analysis information. In Section 5, we evaluate Optgen, state-of-the-art compilers, and the global optimization phase. Finally, Section 6 concludes and discusses future work.

2 Preliminaries and Related Work

The goal of Optgen is to generate a set of local optimization rules that is correct and complete. In this section, we present related work that mostly focuses on assembly level. Along the way, we learn the advantages and drawbacks of working on the IR level rather than on the assembly level.

2.1 Superoptimization Research

During the compilation of a program, the compiler performs many optimizations to improve the resulting code with respect to execution speed, code size, or some

[2] `http://pp.ipd.kit.edu/optgen/`

other criterion. Although the term *optimization* suggests optimal results, modern compilers fail to produce optimal code even for small inputs.

In 1987, Massalin presented a program that can compute the shortest sequence of assembly instructions to realize a given instruction sequence [7]. Since his approach guarantees optimality and the term optimization was already occupied, he called his program *superoptimizer*. The superoptimizer takes a set of instructions and enumerates sequences of them. It then translates the sequence into a boolean expression and compares the resulting minterms with the minterms of the original instruction sequence to decide whether they are equivalent.

Massalin presented two techniques to speed up the superoptimizer. First, he created a set of test inputs and compared the results of the generated sequence and the original one. In his experience, this filters out almost all sequences that are not equivalent. The second speed-up technique is to reject generated sequences that contain a known non-optimal subsequence. With both techniques, the superoptimizer is able to generate sequences of up to 13 instructions in a reasonable amount of time.

In 2002, Joshi et al. presented their superoptimizer *Denali* that allows to find larger optimal sequences [4]. In contrast to Massalin's approach, Denali takes a set of equivalences that should be used to optimize the program. Thus, Denali's task is to find the optimal representation of the input program regarding the given equivalences.

Joshi et al. decided to use E-graphs for a very compact representation of multiple equivalent representations [8]. Denali iteratively applies the given equivalences until the E-graph contains all possible program realizations. Then, Denali constructs a boolean formula that is satisfiable if, and only if, the program can be computed within k cycles. If the formula is satisfiable, Denali can construct a program from the corresponding logical interpretation that uses exactly k cycles. Furthermore, if the formula for $k-1$ is not satisfiable, the previously constructed program uses the minimal number of cycles.

More recently, Schkufza et al. propose to use a Markov chain Monte Carlo sampler to find better versions of a given instruction sequence [9]. Their implementation STOKE sacrifices optimality for the capability to generate optimized sequences of more than 15 instructions. This allows to find sequences that differ algorithmically, which may result in larger speed-ups than an optimal approach that is limited to fewer instructions. In a follow-up paper, Schkufza et al. extended their approach to floating-point arithmetic [10].

2.2 Generators for Peephole Optimizers

Superoptimizers aim to optimize small performance-critical parts of a larger program. In particular, the runtime of a superoptimization run is too high to form an optimization phase of a general-purpose compiler. However, the idea of having some kind of fast superoptimization for arbitrary programs is very attractive.

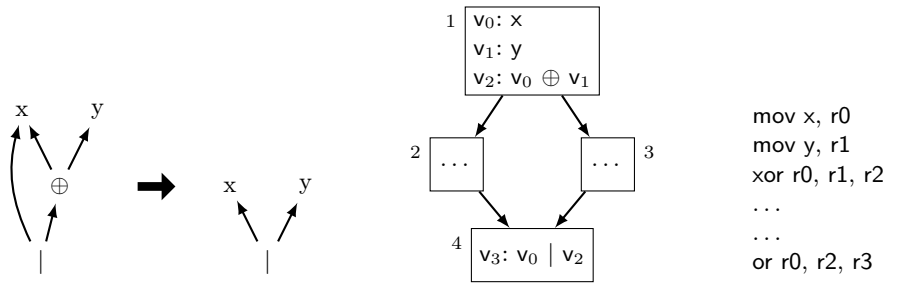
Bansal and Aiken tackle this problem by using training programs to create a peephole optimization database [1]. The compiler's peephole optimization phase can then use a simple look-up to find an applicable optimization for the considered sequence of instructions. Their approach works as follows: First, they compile a set of training programs. Then, the *harvester* extracts all instruction sequences that are candidates for optimizations. The candidates are inserted into a hash table, where the hash is based on the execution of some fixed test inputs. In the second step, they enumerate all instruction sequences up to a given length. For each sequence they perform a look-up in the hash table. If the look-up succeeds, the generated sequence is an optimization candidate for the sequence in the hash table. Thus, they compare both sequences on a larger set of test inputs and finally use a SAT solver to decide whether both sequences are equivalent.

2.3 Generated Optimizations for Intermediate Representations

The tools discussed so far work on the assembly level. This allows to fully leverage the available instruction set and to formulate a precise cost model. However, if we generate optimizations for multiple target architectures, we notice some common optimizations. Following an idiom in compiler design, we should perform these common optimizations on the intermediate representation.

In fact, all compilers come with a set of *local optimizations*. These optimizations consist of small rules that require no global analysis. Thus, the compiler can use these rules at any time, even during construction of an SSA-based IR [2]. In contrast to peephole optimizations, local optimizations have a more global view on the program. They can follow the data dependencies and the sharing of values is not obscured by spilling and other backend phases. Figure 1 illustrates the advantages of working on the IR level. Due to the explicit data dependency, we can model the local optimization as a graph rewrite rule. Figure 1a shows the graph rewrite rule for the optimization $\texttt{x} \mid (\texttt{x} \oplus \texttt{y}) \rightarrow \texttt{x} \mid \texttt{y}$ that can be applied on the IR in Figure 1b. However, a peephole optimizer cannot apply this rule on the assembly level, since the instructions of basic blocks 2 and 3 occur between the instructions that belong to the optimization rule.

Currently, the compiler's local optimizations are handcrafted and reflect the knowledge of the compiler developers. Thus, the optimization rules guarantee neither correctness nor completeness. Regarding correctness, the ALIVe tool [6] demonstrates a possible approach to verify local optimizations. A promising solution to get the completeness guarantee is to port the idea of a generator for peephole optimizations to the IR level. So far, there is only little research regarding this idea. For instance, Tate et al. pick up the ideas of Denali and apply them to their intermediate representation [12]. However, to the best of our knowledge, there is no approach that tackles the systematic generation of local optimizations.



(a) Graphical representation of the rule `x | (x ⊕ y)` → `x | y`.

(b) Program in SSA form.

(c) Generated assembly code.

Fig. 1. The local optimization of Figure 1a can be applied on the IR of Figure 1b. However, on the assembly level of Figure 1c, the generated code of basic blocks 2 and 3 prevent the application of the corresponding peephole optimization.

3 Rule Generation

In this chapter, we present and discuss our tool Optgen that generates local optimizations. When creating Optgen, our idea was to generate *all* local optimizations up to a given cost limit. The main challenge for this purpose is the handling of constants: Since a 32-bit architecture has 2^{32} constants, enumerating all constant folding rules for a binary operation would result in 2^{64} rules. Obviously, this cannot be accomplished in a reasonable amount of time. In the following, we present the general design of Optgen and explain how we tackle the large amount of available constants.

3.1 General design of Optgen

Optgen's task is to generate all local optimizations up to a given cost limit. Thus, Optgen takes the considered operations and their costs, as well as the cost limit, as input parameters. Furthermore, the user must specify the bit width of the operations. It then generates the local optimizations and outputs them in textual and graphical form. Furthermore, it can generate a test suite that can be used to find missing optimizations in existing compilers. Currently, Optgen supports the unary integer operations $\sim$ and `-`, as well as the binary integer operations `+`, `&`, `|`, `-` and $\oplus$. However, adding a new operation only requires a mapping to the SMT solver and a method to evaluate the operation for constant operands.

We use Figure 2 and a running example to demonstrate the work flow of Optgen: We want Optgen to generate all local optimizations up to cost 2 for the 8-bit operations `&` and `|`, which both have cost 1. Before Optgen starts the actual generation, it creates a number of random test inputs. Later, we will use these random tests to compute a *semantic hash* for each expression. The

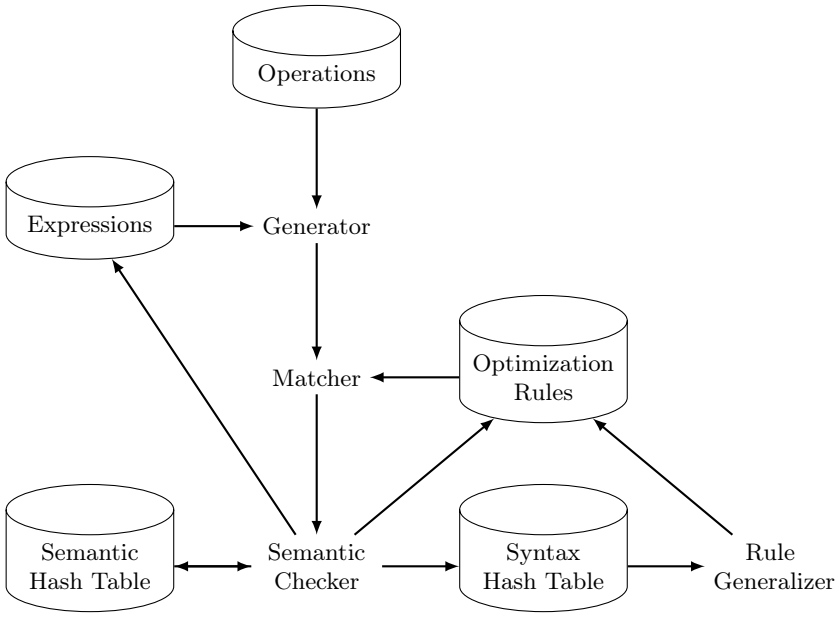


Fig. 2. General design of OPTGEN.

underlying idea is that if two expressions evaluate to different values for the test inputs, the considered expressions cannot be semantically equivalent.

OPTGEN now generates all expressions for each cost. For cost 0 the *generator* generates the variable `x` and the constants `0` to `255`. The generator passes each of these expressions to the *matcher*. The matcher checks whether we have already found an optimization rule that applies to the given expression. Since we have found no optimization rule yet, the matcher passes the expression to the *semantic checker*. The semantic checker computes the semantic hash and looks up a list of possibly equivalent expressions in the *semantic hash table*. Assuming a perfect hash function, the lookup finds no such expression in our example. Thus, the semantic checker inserts the expression into the list of expression as well as into the *semantic hash table*.

Before OPTGEN generates the expressions with cost 1, it realizes that we have a binary operation with cost 1 and introduces a new variable `y`. Then, it starts the actual generation process by applying the available operations to the already generated expressions. The first generated expression is `x & x`. Since we have found no optimization rule yet, the matcher passes the expression to the semantic checker. The semantic checker now computes the semantic hash and looks up a list of possibly equivalent expressions. Assuming a perfect hash function, our list only contains the expression `x`. The semantic checker now uses an SMT solver to determine whether both expressions are equivalent. In our case, the expressions are equivalent. Since `x` is cheaper than `x & x`, OPTGEN creates a

new optimization rule `x & x → x` and inserts it into the list of rules. In the next step, OPTGEN creates the optimization rule `x & 0 → 0` in a similar fashion.

The next generated expression is `x & 1`. Again, the matcher finds no existing optimization rule and passes the expression to the semantic checker that looks up possibly equivalent expressions in the semantic hash table. Let us assume the hash table lookup finds some candidate, e.g., the expression `1`. In this case, the following SMT check fails. Thus, we insert the expression `x & 1` into the list of candidates for the computed hash value.

When generating expressions with cost 2, another interesting case occurs: Processing the expression `(x & y) & 0`, the matcher finds the applicable optimization rule `x & 0 → 0`. In this case, we skip the generated expression. Otherwise, the semantic checker would create the optimization rule `(x & y) & 0 → 0` that is subsumed by the existing optimization rule `x & 0 → 0`.

3.2 Handling constants

For our running example, OPTGEN generates many similar constant folding rules like `1 & 2 → 0` and `1 & 3 → 1`. Returning each of these rules to the user is inconvenient. Instead, the user is interested in a single constant folding rule for each operation. Thus, OPTGEN provides a *rule generalizer* that tries to generalize the generated rules. In our running example, we want to create a rule `c0 & c1 → eval(c0 & c1)`, where `c0` and `c1` are *symbolic constants* and `eval` performs constant folding.

Before we can start the generalization, we need to find sets of syntactically equivalent rules. We solve this problem by computing a *syntax hash* for each expression that only depends on the structure of the expression, i.e., ignoring the values of the constants. Thus, we can use a *syntax hash table* to efficiently find syntactically equivalent rules. As shown in Figure 2, the semantic checker is responsible for filling the syntax hash table with rules that contain constants.

Given a set of syntactically equivalent rules, the rule generalizer first tries to find expressions that compute the constants of the right-hand side using the constants of the left-hand side. Currently, it does this by considering the already enumerated expressions. In our running example, we have one constant on the right-hand side and two constants on the left-hand side. Hence, the rule generalizer searches a function `f(x,y)` such that `f(1,2) = 0`, `f(1,3) = 1` and so on. If it finds an appropriate function, it creates the corresponding rule and checks its correctness using an SMT solver. If the check succeeds, the rule generalizer found a rule that supersedes the considered rules. Otherwise, it continues the search for an appropriate function. In our running example, the rule generalizer finds `f(x,y) = x & y` and verifies the resulting rule `c0 & c1 → eval(c0 & c1)`.

An interesting situation occurs for the rules `(x | 2) & 1 → x & 1`, `(x | 1) & 2 → x & 2` and so on. In general, the optimization `(x | c1) & c2 → x & c2` is not valid. However, it becomes valid if `c1` and `c2` are bitwise disjoint. OPTGEN tackles such cases by using *conditional rules*. By definition, a conditional rule is valid if the corresponding condition evaluates to `true`. Furthermore, the condition must

be a function that only depends on the symbolic constants that appear on the left-hand side of the rule.

The rule generalizer solves the problem of finding a condition by searching a *condition expression* that evaluates to `0` if, and only if, the condition holds. Finding a condition expression is similar to finding the computations for constants on the right-hand side of a rule: We simply check the already enumerated expressions for an appropriate one. Since the enumerated expressions are sorted according to their costs, simpler condition expressions are considered before complex ones. For our example, OPTGEN finds the appropriate condition expression `c1 & c2` and creates the conditional rule `(c1 & c2) == 0` ⇒ `(x | c1) & c2` → `x & c2`.

The main advantage of symbolic constants is their independence of the bit width. Thus, the SMT solver can verify the local optimization rule `(c1 & c2) == 0` ⇒ `(x | c1) & c2` → `x & c2` for 8 bits as well as for 32 bits. If we generalize every non-trivial set of syntactically equivalent rules, we can easily extend the verification from 8 bits to 32 bits. For the remaining rules with particular constants, OPTGEN expands the 8-bit constants to 32 bits by padding the most significant or least significant bits with zeros or ones and checks all four resulting rules using an SMT solver. In our experience, this approach is sufficient to find the corresponding 32-bit rule. Thus, we can generate all local optimizations for 8 bits, extend the resulting rules to 32 bits, and verify them for the latter bit width. Whether the rule set of the extended bit width is also complete depends on the operations and the bit width used during generation. A proven approach to achieve completeness is to increase the generation bit width until the generated rule set for the extended bit width remains unchanged. For instance, generating all 3-bit rules with cost limit 3 for the bitwise operations $\sim$, `&`, `|` and $\oplus$ creates the same 32-bit rule set as generating all 4-bit rules.

Currently, OPTGEN can only generalize rules if the required condition expression and computations of the symbolic constants on the right-hand side are enumerated expressions. A more general solution would use superoptimization techniques to find the appropriate expressions. The basic idea is that the existing optimization rules define partial functions for the condition expression and computations of the symbolic constants that appear on the right-hand side of the generalized rule. These functions are partial, because we skip expressions that are matched by existing optimization rules. For instance, the generalized rule `(c1 & c2) == 0` ⇒ `(x | c1) & c2` → `x & c2` covers the optimization rule `(x | 1) & 0` → `x & 0`. However, we will never create the latter one because we can apply `x & 0` → `0` on its left-hand side.

Based on the partial functions, we can use superoptimization techniques to find the appropriate expressions. This may require to limit the cost of each expression to guarantee termination. If we found an expression for each partial function, we can use the SMT solver the verify the corresponding rule. If the SMT check fails, the SMT solver can generate a counterexample for these expressions. Thus, we can use this counterexample to refine our partial functions and continue the search for appropriate expressions.

3.3 Performance Tuning

In this section, we present some speed-up techniques implemented in Optgen.

The key insight regarding performance is that SMT checks are slow. Thus, we need to avoid them if possible. The main source of unnecessary SMT checks are collisions in the semantic hash table. In this case, a created expression could have multiple candidates of equivalent expressions that are compared via SMT checks. However, at most one of them can be equivalent to the expression at hand.

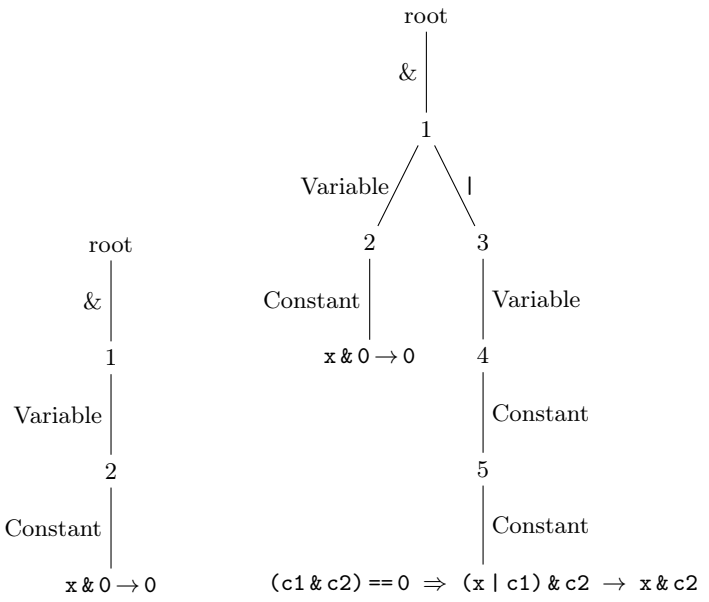
Our solution to this problem are *witnesses*: A set of test inputs such that each pair of expressions of the hash table bucket evaluates to a different result for at least one witness. Since each hash table bucket has it own set of witnesses, these sets are usually very small. For a new expression with multiple candidates, we first evaluate the witnesses and compare the results with the results of the available candidates. By definition, at most one candidate can compute the same results for all witnesses. Thus, we need at most one SMT check per generated expression. Whenever this SMT check fails, we insert the new expression into the hash table bucket. Hence, we need a new witness to distinguish the two expressions. Fortunately, the SMT solver also generates a counterexample as a result of the failing SMT check. Thus, we simply use this counterexample as a new witness for the checked expressions.

Another performance-critical task is to check the applicability of existing rules to expressions. Optgen performs such checks for each generated expression. Furthermore, Optgen uses these checks to identify rules that are covered by more general ones. For an efficient matching, we use an n-ary tree structure that can be indexed by the available operation types. If we find a new optimization rule, we traverse the nodes of the left-hand side in preorder and insert the corresponding nodes into the tree. The created leaf contains a reference to the inserted rule. Figure 3 shows the search tree after inserting the rules `x & 0` $\rightarrow$ `0` and `(c1 & c2) == 0` $\Rightarrow$ `(x | c1) & c2` $\rightarrow$ `x & c2`.

Assume we want to find a rule that matches `(x | 3) & 1`. In order to use our data structure, we process our expression in preorder. The type of the first operation is `&`. The matcher now descends into every child that corresponds to a type that matches `&`. Thus, it visits node 1 and proceeds with the `|` node of the expression. Since both existing children of the search tree are labeled with matching types, we must visit them both.

Let us assume the matcher first descends into node 2. Since the variable matches the whole subexpression `x | 3`, the matcher skips the corresponding nodes of the expression and proceeds with the constant `1`. In the next step, the matcher reaches the leaf that contains the rule `x & 0` $\rightarrow$ `0`. Thus, it tries to apply the rule to the expression. Unfortunately, the constant `0` does not match the constant `1`.

The matcher continues its search by descending from node 1 to node 3. Since the path to the second rule fits our expression, the matcher tries to apply the rule `(c1 & c2) == 0` $\Rightarrow$ `(x | c1) & c2` $\rightarrow$ `x & c2`. This time the condition of the rule is not fulfilled. Thus, the matcher continues its search. Since the matcher



(a) Tree data structure after inserting `x & 0 → 0`.

(b) Tree data structure after further inserting `(c1 & c2) == 0 ⇒ (x | c1) & c2 → x & c2`.

Fig. 3. Tree data structure to find applicable rules for an expression. Each inner node can be indexed by the available operations types. For an insert or lookup operation the operation types of the expression are considered in preorder.

has already processed the interesting paths of the search tree, it stops and reports that no matching rule exists.

3.4 Applications of OPTGEN

OPTGEN supports the compiler developer by providing an optimization test suite. The test suite helps to find missing optimizations of the developed compiler. Furthermore, it identifies optimization applications that should be prevented by the compiler in case of shared subexpressions. Figure 4 shows such a situation: Applying the local optimization `-((x-y)+z)` $\rightarrow$ `y-(x+z)` increases the global costs, because of the shared subexpression `(x-y)+z`. However, if the value `(x-y)+z` is already present the optimization is worthwhile. A possible solution to this problem would be to conservatively prevent optimizations in case of shared subexpressions.

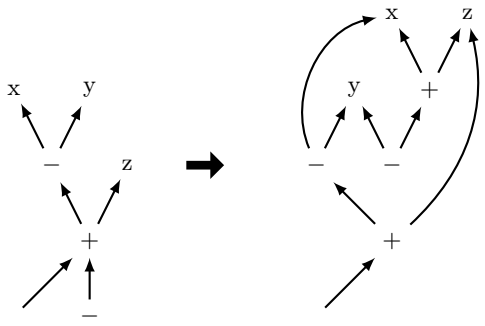


Fig. 4. The compiler should prevent the application of the local optimization rule `-((x-y)+z)` $\rightarrow$ `y-(x+z)`, since the subexpression `(x-y)+z` has another user.

4 Combining Local Optimizations with Global Analyses

Local optimizations have the advantage that they do not require any prior analysis. Thus, they can be applied at any time, e.g., directly after the construction of an operation. In our experience, it is also worthwhile to have a compiler phase that applies all local optimizations until it reaches a fixpoint. During this phase, we can provide analysis information that improves the existing local optimizations.

In the following, we present two analyses that allow a compact and powerful implementation of the local optimizations generated by OPTGEN. As we will see, the generated local optimization rules with symbolic constants are crucial to our approach. We start with a simple optimization rule to motivate our idea: `(x+2)&1` $\rightarrow$ `x&1`. OPTGEN covers this rule by the generalized

rule `((c0 | -c0) & c1) == 0` $\Rightarrow$ `(x + c0) & c1` $\rightarrow$ `x & c1`. The condition ensures that adding the constant `c0` only influences bits that are masked out by the constant `c1`. Our plan is to stepwise relax the condition to apply this optimization in even more cases.

4.1 Constant-Bit Analysis

Our first insight to improve the optimization `(x + 2) & 1` $\rightarrow$ `x & 1` is that the second operand of the addition can be an arbitrary expression as long as the rightmost bit is not set. For instance, `(x + (y & 42)) & 1` $\rightarrow$ `x & 1` is also a valid optimization rule.

This motivates us to implement a *constant-bit analysis* that indicates bits that are always set or not set, respectively. For the expression `y & 42`, the analysis computes `00?0 ?0?0` for the rightmost 8 bits, where `0` indicates bits that are guaranteed to be zero and `?` indicates bits that are not constant. Similarly, `y | 42` results in `??1? 1?1?`. The bit information is stored in two bit vectors. The bit vector `zeros` contains a cleared bit if the corresponding bit is guaranteed to be cleared and a set bit otherwise. Likewise, the bit vector `ones` contains a set bit if the corresponding bit is guaranteed to be set and a cleared bit otherwise.

The Constant-bit analysis is a forward data flow analysis that generalizes sparse conditional constant propagation [13]. Since the data-flow analysis is optimistic, it initializes the `zeros` bit vector with zeros and the `ones` bit vector with ones. Then, it applies the transfer function until the analysis reaches the fixpoint.

Some of the transfer functions are straight-forward. For constants we set the bit vectors according to the bits of the constants. For loads from memory we set all bits of the `zeros` vector and clear the bits of the `ones` vector. The bitwise operations `&` and `|` can be transformed by applying the same operation to corresponding bit vectors of the operands. For instance, `ones(x & y) = ones(x) & ones(y)`. For the bitwise complement $\sim$ we must apply the operation to the other bit vector of the operand: `ones(`$\sim$`x) =` $\sim$`zeros(x)`, `zeros(`$\sim$`x) =` $\sim$`ones(x)`. Contrarily, the transfer function for the exclusive or $\oplus$ is more complex:
`ones(x` $\oplus$ `y) = (ones(x) &` $\sim$`zeros(y)) | (ones(y) &` $\sim$`zeros(x))`,
`zeros(x` $\oplus$ `y) = (zeros(x) &` $\sim$`ones(y)) | (zeros(y) &` $\sim$`ones(x))`.

Transfer functions for arithmetic operations also reuse the operation itself. For the addition `x + y`, we first add the `ones` and `zeros` of the operands: `vo = ones(x) + ones(y)` and `vz = zeros(x) + zeros(y)`. Then, we compute bit vectors that indicate which bits are not constant: `xnc = ones(x)` $\oplus$ `zeros(x)`, `ync = ones(y)` $\oplus$ `zeros(y)` and `vnc = vo` $\oplus$ `vz`. This allows us to determine the non-constant bits of the result: `nc = xnc | ync | vnc`. Finally, we can compute the constant-bit information of `x + y`: `ones(x + y) = vz &` $\sim$`nc` and `zeros(x + y) = vz | nc`. Due to the use of the add operation, we handle carry bits correctly. For instance, `00?? 1110 + 10?? 1?10` results in `1??? 1?00`.

A typical use case for constant-bit information is to determine the equivalence of multiple operations. If the operands of an addition have disjoint bits

set, we can also use a `|` or $\oplus$ operation. Thus, the optimizer may apply local optimizations that are valid for the `|` operation but not in general for the addition.

4.2 Don't Care Analysis

The second insight to improve the optimization `(x+2)&1` $\rightarrow$ `x&1` is that due to the `&`, we only care for the least significant bit of the sum. The don't care analysis provides exactly this information: Its result is a bit vector that indicates relevant (1) and irrelevant (0) bits [11]. This allows to specify the more compact optimization rule `care(x+2) == 1` $\Rightarrow$ `x+2` $\rightarrow$ `x`, which also covers `(x+2)|`$\sim$`1` $\rightarrow$ `x|`$\sim$`1`.

In contrast to the constant-bit analysis, the don't care analysis is a backward data-flow analysis. At the beginning of the analysis, all bits are set to irrelevant. Since the transfer functions of `return` or `store` operations always care for their operands, they create some initial relevant bits. These bits will then propagate through the program until the analysis reaches its fixpoint.

There are several expressions that can create more irrelevant bits for at least one of their operands: `x|1`, `x*2`, `x&2`, and so on. In most cases, the don't care analysis uses known bits from one operand and derives irrelevant bits for the other one. Thus, we extend the don't care analysis to consider the constant-bit information. In consequence, if we use constant-bit information to gain precision, we must care for the bits that provided this constant-bit information.

4.3 Generalizing the Optimization Rules

Starting with the optimization rule `(x+2)&1` $\rightarrow$ `x&1`, we manually derived the optimization `((zeros(y)|-zeros(y))&care(x+y))==0` $\Rightarrow$ `x+y` $\rightarrow$ `x`. The generalized rule encapsulates the essential optimization, while using the global analysis information to check whether the rule can be applied. This allows to perform the optimization even in complex scenarios.

The presented analyses allow an even more compact optimization: The creation of *occult constants* [11]. This optimization can be performed if all relevant bits are known to be constant: `((zeros(x)`$\oplus$`ones(x))&care(x))==0` $\Rightarrow$ `x` $\rightarrow$ `eval(zeros(x)&care(x))`. For instance, the constant `2` of the expression `(x+2)&1` is an occult constant that can be optimized to zero. However, in a larger program the constant `2` can have other users that render more bits relevant.

Currently, OPTGEN does not derive local optimization rules that use the presented analysis information. The constant-bit analysis would require another analysis that determines whether we need a conservative approximation of the set or cleared bits. For the don't care analysis, we would just need to perform the analysis for the expressions of the rule. Furthermore, we would need to adapt the generated formulas for the SMT solver. Similar to the generation of symbolic rules discussed in Section 3.2, superoptimization techniques could be helpful to find appropriate conditions for the derived rules.

5 Evaluation

In this section, we evaluate OPTGEN's runtime and compare its generated optimizations with state-of-the art compilers. All measurements are performed on an Intel Core i7-3770 3.40 GHz with 16 GB RAM. The machine runs a 64-bit Ubuntu 14.04 LTS distribution that uses the 3.13.0-37-generic version of the Linux kernel. For OPTGEN, we use Z3 4.3.1 as SMT solver [3].

5.1 OPTGEN Runtime

In order to evaluate the runtime of the generation of local optimizations, we run OPTGEN in multiple configurations. All runs include the unary operations $\sim$ and `-`, as well as the binary operations `+`, `&`, `|`, `-` and $\oplus$. Table 1 shows the different configurations as well as the resulting runtime and maximum memory usage for each configuration. The configurations differ in the used bit width for the rule generation, in the number of involved operations and in the usage of constants.

Operations	Bit width		Constants	Runtime	Memory Usage
	Generation	Verification			
2	8	32	✓	6 h 7 min 0 s	1 046 568 kB
2	8	32	×	1 s	7456 kB
2	32	32	×	6 min 21 s	19 892 kB
3	8	32	×	36 s	17 900 kB
4	8	32	×	8 h 27 min 16 s	686 104 kB

Table 1. Runtime and maximum memory usage of OPTGEN for different configurations.

For a fixed number of operations, the results suggest two aspects that heavily influence the runtime: The use of constants and the bit width used during the rule generation. We already argued that the use of constants increases the number of generated rules and, thus, the runtime. However, the bit width only influences the SMT checks. Since the generation for two 8-bit operations is significantly faster than the generator for two 32-bit operations, the SMT solver does not scale very well with increasing bit width. Consequently, using different bit widths for generation and verification dramatically improves the runtime.

5.2 Testing State-of-the-Art Compilers

In order to give valuable feedback to compiler engineers, we let OPTGEN generate an optimization test suite. This test suite includes a test for each generated optimization. Using these tests, it lets the compiler of interest generate x86-64 assembly and then counts the number of generated arithmetic instructions.

If the compiler generates more instructions than expected, we found a missing optimization for the compiler.

We use the run of Optgen with two operations and constants to test state-of-the-art compilers. Table 2 shows the missing optimizations for GCC 4.9, LLVM 3.5, and ICC 15. In total, Optgen found 63 optimizations that are missing in at least one of the compilers. The optimizations include rules without constants (20.), rules with symbolic constants (54.), and rules with particular constants (26.). Since Optgen currently considers only generated expressions for conditional rules, the conditions of the rules 16, 21, 22, and 59 are created by hand.

	Optimization	Compiler		
		LLVM	GCC	ICC
1.	`-∼x → x+1`	✓	✓	×
2.	`-(x&0x80000000) → x&0x80000000`	×	✓	×
3.	`∼-x → x-1`	✓	✓	×
4.	`x+∼x → 0xFFFFFFFF`	✓	✓	×
5.	`x+(x&0x80000000) → x&0x7FFFFFFF`	×	×	×
6.	`(x|0x80000000)+0x80000000 → x&0x7FFFFFFF`	✓	×	×
7.	`(x&0x7FFFFFFF)+(x&0x7FFFFFFF) → x+x`	✓	✓	×
8.	`(x&0x80000000)+(x&0x80000000) → 0`	✓	✓	×
9.	`(x|0x7FFFFFFF)+(x|0x7FFFFFFF) → 0xFFFFFFFE`	✓	✓	×
10.	`(x|0x80000000)+(x|0x80000000) → x+x`	✓	✓	×
11.	`x&(x+0x80000000) → x&0x7FFFFFFF`	✓	×	×
12.	`x&(x|y) → x`	✓	✓	×
13.	`x&(0x7FFFFFFF-x) → x&0x80000000`	×	×	×
14.	`-x&1 → x&1`	×	✓	×
15.	`(x+x)&1 → 0`	✓	✓	×
16.	`is_power_of_2(c1) && c0&(2*c1-1)==c1-1 ⇒ (c0-x)&c1 → x&c1`	×	×	×
17.	`x|(x+0x80000000) → x|0x80000000`	✓	×	×
18.	`x|(x&y) → x`	✓	✓	×
19.	`x|(0x7FFFFFFF-x) → x|0x7FFFFFFF`	×	×	×
20.	`x|(x⊕y) → x|y`	✓	×	×
21.	`((c0|-c0)&∼c1)==0 ⇒ (x+c0)|c1→x|c1`	✓	×	✓
22.	`is_power_of_2(∼c1) && c0&(2*∼c1-1)==∼c1-1 ⇒ (c0-x)|c1 → x|c1`	×	×	×
23.	`-x|0xFFFFFFFE → x|0xFFFFFFFE`	×	×	×
24.	`(x+x)|0xFFFFFFFE → 0xFFFFFFFE`	✓	✓	×
25.	`0-(x&0x80000000) → x&0x80000000`	×	✓	×
26.	`0x7FFFFFFF-(x&0x80000000) → x|0x7FFFFFFF`	×	×	×
27.	`0x7FFFFFFF-(x|0x7FFFFFFF) → x&0x80000000`	×	×	×
28.	`0xFFFFFFFE-(x|0x7FFFFFFF) → x|0x7FFFFFFF`	×	×	×
29.	`(x&0x7FFFFFFF)-x → x&0x80000000`	×	×	×

Table 2: Missing local optimizations of state-of-the-art compilers. A ✓ indicates that the corresponding optimization is supported, whereas a × indicates a missing optimization.

	Optimization	Compiler		
		LLVM	GCC	ICC
30.	$x \oplus (x + 0x80000000) \rightarrow 0x80000000$	✓	×	×
31.	$x \oplus (0x7FFFFFFF - x) \rightarrow 0x7FFFFFFF$	×	×	×
32.	$(x + 0x7FFFFFFF) \oplus 0x7FFFFFFF \rightarrow -x$	×	×	×
33.	$(x + 0x80000000) \oplus 0x7FFFFFFF \rightarrow \sim x$	✓	✓	×
34.	$-x \oplus 0x80000000 \rightarrow 0x80000000 - x$	×	×	×
35.	$(0x7FFFFFFF - x) \oplus 0x80000000 \rightarrow \sim x$	×	✓	×
36.	$(0x80000000 - x) \oplus 0x80000000 \rightarrow -x$	×	✓	×
37.	$(x + 0xFFFFFFFF) \oplus 0xFFFFFFFF \rightarrow -x$	✓	✓	×
38.	$(x + 0x80000000) \oplus 0x80000000 \rightarrow x$	✓	✓	×
39.	$(0x7FFFFFFF - x) \oplus 0x7FFFFFFF \rightarrow x$	×	×	×
40.	$x - (x \,\&\, c) \rightarrow x \,\&\, \sim c$	✓	✓	×
41.	$x \oplus (x \,\&\, c) \rightarrow x \,\&\, \sim c$	✓	✓	×
42.	$\sim x + c \rightarrow (c - 1) - x$	✓	✓	×
43.	$\sim(x + c) \rightarrow \sim c - x$	✓	×	×
44.	$-(x + c) \rightarrow -c - x$	✓	✓	×
45.	$c - \sim x \rightarrow x + (c + 1)$	✓	✓	×
46.	$\sim x \oplus c \rightarrow x \oplus \sim c$	✓	✓	×
47.	$\sim x - c \rightarrow \sim c - x$	✓	✓	×
48.	$-x \oplus 0x7FFFFFFF \rightarrow x + 0x7FFFFFFF$	×	×	×
49.	$-x \oplus 0xFFFFFFFF \rightarrow x - 1$	✓	✓	×
50.	$x \,\&\, (x \oplus c) \rightarrow x \,\&\, \sim c$	✓	✓	×
51.	$-x - c \rightarrow -c - x$	✓	✓	×
52.	$(x \mid c) - c \rightarrow x \,\&\, \sim c$	×	×	×
53.	$(x \mid c) \oplus c \rightarrow x \,\&\, \sim c$	✓	✓	×
54.	$\sim(c - x) \rightarrow x + \sim c$	✓	×	×
55.	$\sim(x \oplus c) \rightarrow x \oplus \sim c$	✓	✓	×
56.	$\sim c0 == c1 \Rightarrow (x \,\&\, c0) \oplus c1 \rightarrow x \mid c1$	✓	✓	×
57.	$-c0 == c1 \Rightarrow (x \mid c0) + c1 \rightarrow x \,\&\, \sim c1$	×	×	×
58.	$(x \oplus c) + 0x80000000 \rightarrow x \oplus (c + 0x80000000)$	✓	✓	×
59.	$((c0 \mid -c0) \,\&\, c1) == 0 \Rightarrow (x \oplus c0) \,\&\, c1 \rightarrow x \,\&\, c1$	✓	✓	×
60.	$(c0 \,\&\, \sim c1) == 0 \Rightarrow (x \oplus c0) \mid c1 \rightarrow x \mid c1$	✓	×	×
61.	$(x \oplus c) - 0x80000000 \rightarrow x \oplus (c + 0x80000000)$	✓	✓	×
62.	$0x7FFFFFFF - (x \oplus c) \rightarrow x \oplus (0x7FFFFFFF - c)$	×	×	×
63.	$0xFFFFFFFF - (x \oplus c) \rightarrow x \oplus (0xFFFFFFFF - c)$	✓	✓	×
Sum		23	27	62

Table 2: Missing local optimizations of state-of-the-art compilers. A ✓ indicates that the corresponding optimization is supported, whereas a × indicates a missing optimization.

As discussed in Section 4.3, we further tested whether the compilers prevent optimizations if all subexpressions are shared. Since such cases do not appear

with two operations, we used the optimizations with three operations but without constants. Table 3 shows all cases, where at least one compiler increases the cost by applying the corresponding optimization. For instance, the first optimization $\sim$(x | $\sim$y) $\rightarrow$ $\sim$x & y is supported by GCC and LLVM. However, only GCC prevents the application if the subexpression (x | $\sim$y) is used by another operation.

	Optimization	Compiler		
		LLVM	GCC	ICC
1.	$\sim$(x | $\sim$y) $\rightarrow$ $\sim$x & y	×	✓	
2.	$\sim$(x & $\sim$y) $\rightarrow$ $\sim$x | y	×	✓	
3.	(x + x) & (y + y) $\rightarrow$ (x & y) + (x & y)	×		
4.	(x + x) | (y + y) $\rightarrow$ (x | y) + (x | y)	×		
5.	(x & y) | (z & y) $\rightarrow$ y & (x | z)	✓	×	✓
6.	x - ((x - y) + (x - y)) $\rightarrow$ y + (y - x)		✓	×
7.	(x - y) - (x + z) $\rightarrow$ -(y + z)	✓	✓	×
8.	((x - y) + (x - y)) - x $\rightarrow$ x - (y + y)	✓	✓	×
9.	(x + x) $\oplus$ (y + y) $\rightarrow$ (x $\oplus$ y) + (x $\oplus$ y)	×		
10.	(x & y) $\oplus$ (z & y) $\rightarrow$ y & (x $\oplus$ z)	✓	×	✓

Table 3. State-of-the-art compilers apply optimization rules even if the operands are shared. If the compiler supports the optimization ✓/× indicates whether the compiler prevents the optimization in case of shared operands. If the compiler does not support the optimization the item is left blank.

5.3 Global Optimization Phase

In Section 4, we claimed that it is worthwhile to perform local optimizations until a fixpoint is reached. We used the libFirm compiler [5] and the SPEC CINT2000 benchmark to prove our claim. Table 4 shows the number of executed instructions of the generated binaries. On average, the application of local optimizations until the fixpoints reduces the number of executed instructions by 3.22 %.

Furthermore, we evaluated the effect of the constant-bit and don't care analyses. Table 5 shows that using the analyses further improved the overall performance by 0.64 %. The enabled analyses achieve their best improvement for the 186.crafty benchmark that contains a lot of bit operations. Here, the analyses discover an occult constant that results in an improvement by 3.14 %.

6 Conclusion and Future Work

In this paper, we presented the local optimization generator Optgen. In contrast to generators for peephole optimizers, Optgen generates optimization rules that

Benchmark	Without Local Phase	With Local Phase	$\frac{\text{Without Local Phase}}{\text{With Local Phase}}$
164.gzip	306,800,522,532	290,253,056,191	105.70 %
175.vpr	215,443,006,054	203,496,140,283	105.87 %
176.gcc	150,470,998,502	149,081,148,091	100.93 %
181.mcf	48,034,571,679	48,924,041,098	98.18 %
186.crafty	192,013,840,675	184,861,683,227	103.87 %
197.parser	313,187,450,212	291,055,655,200	107.60 %
253.perlbmk	1,147,112,186	1,106,164,617	103.70 %
254.gap	220,455,529,344	216,480,669,077	101.84 %
255.vortex	329,083,783,116	311,764,008,973	105.56 %
256.bzip2	285,176,110,773	278,769,705,624	102.30 %
300.twolf	293,847,320,971	293,190,467,622	100.22 %
Average			103.22 %

Table 4. Effect of local optimizations phase. The table compares the number of executed instructions of the generated code with and without an optimization phase that applies local optimizations until a fixpoint is reached.

Benchmark	Without Analyses	With Analyses	$\frac{\text{Without Analyses}}{\text{With Analyses}}$
164.gzip	290,253,056,191	285,201,958,027	101.77 %
175.vpr	203,496,140,283	203,495,172,137	100.00 %
176.gcc	149,081,148,091	146,088,405,963	102.05 %
181.mcf	48,924,041,098	48,969,263,250	99.91 %
186.crafty	184,861,683,227	179,226,675,963	103.14 %
197.parser	291,055,655,200	291,053,327,116	100.00 %
253.perlbmk	1,106,164,617	1,106,163,127	100.00 %
254.gap	216,480,669,077	216,345,138,043	100.06 %
255.vortex	311,764,008,973	311,764,025,981	100.00 %
256.bzip2	278,769,705,624	278,229,736,806	100.19 %
300.twolf	293,190,467,622	293,190,431,795	100.00 %
Average			100.64 %

Table 5. Effect of constant-bit and don't care analyses. The table compares the number of executed instructions of the generated code with and without the constant-bit and don't care analyses.

work on the IR level. This allows a more abstract view on the program behavior than working on assembly level.

A unique feature of Optgen is its full support of constants. This includes the generalization of syntactically equivalent rules to a rule with symbolic constants. Furthermore, we demonstrated that it is sufficient to generate rules for a small bit width and later extend them to a larger bit width. Together, these techniques allow the efficient generation of all rules that involve constants.

We further generalized the generated rules of Optgen by using the analysis information of the constant-bit and don't care analyses. This compacts the rule specification and allows to apply the local optimizations in more cases. Using the SPEC CINT2000 benchmark, we obtain an reduction of the executed instructions by up to 3.14 %, when using the analysis information.

We used Optgen to find unsupported optimizations in the state-of-the-art compilers GCC, ICC and LLVM. For these compilers, we identified more than 50 missing optimizations that involve at most two operations. Furthermore, we showed that compilers should prevent the application of some optimization rules due to shared subexpressions. For the state-of-the-art compilers, we identified ten optimizations that are applied in such scenarios.

During the development of Optgen, we identified three interesting research topics for future work. The first idea is the use of superoptimization techniques to derive appropriate conditions during the rule generalization as discussed at the end of Section 3.2. Furthermore, such techniques can be used to find optimal implementations for the transfer functions of the constant-bit analysis. The second idea is to extend Optgen to automatically derive condition expressions that are based on constant-bit and don't care information as discussed at the end of Section 4.3. The third research topic concerns the sharing of subexpressions. Here, algorithms that reassociate existing expression to allow the application of local optimizations or to improve the sharing with existing subexpressions would be worthwhile for state-of-the-art compilers.

Acknowledgments We thank Christoph Mallon and Manuel Mohr for many fruitful discussions and valuable advices, and the anonymous reviewers for their helpful comments. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 394–403. ASPLOS XII, ACM, New York, NY, USA (2006)
2. Braun, M., Buchwald, S., Hack, S., Leißa, R., Mallon, C., Zwinkau, A.: Simple and efficient construction of static single assignment form. In: Jhala, R., Bosschere, K. (eds.) Compiler Construction. Lecture Notes in Computer Science, vol. 7791, pp. 102–122. Springer Berlin Heidelberg (2013)

3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
4. Joshi, R., Nelson, G., Randall, K.: Denali: A goal-directed superoptimizer. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 304–314. PLDI '02, ACM, New York, NY, USA (2002)
5. libFirm – The FIRM intermediate representation library, `http://libfirm.org`
6. Lopes, N., Menendez, D., Nagarakatte, S., Regehr, J.: ALIVe: Automatic LLVM InstCombine Verifier, `http://blog.regehr.org/archives/1170`
7. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems. pp. 122–126. ASPLOS II, IEEE Computer Society Press, Los Alamitos, CA, USA (1987)
8. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM 27(2), 356–364 (Apr 1980)
9. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 305–316. ASPLOS '13, ACM, New York, NY, USA (2013)
10. Schkufza, E., Sharma, R., Aiken, A.: Stochastic optimization of floating-point programs with tunable precision. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 53–64. PLDI '14, ACM, New York, NY, USA (2014)
11. Seltenreich, A.: Minimizing bit width using data flow analysis in libfirm (Feb 2013)
12. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 264–276. POPL '09, ACM, New York, NY, USA (2009)
13. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13(2), 181–210 (Apr 1991)