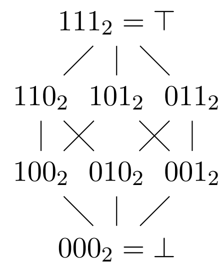
Minimizing bit width using data flow analysis in libFirm

Andreas Seltenreich

January 24, 2013



Studienarbeit am Karlsruhe Institute of Technology, Fakultät für Informatik, Institut für Programmstrukturen und Datenorganisation.

Verantwortlicher Mitarbeiter: Prof. Gregor Snelting
Betreuender Mitarbeiter: Dipl.-Inform. Andreas Zwinkau

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

....
(Andreas Seltenreich)

Contents

1 Introduction **6**

2 Preliminaries **6**
2.1 Notation . . . 6

3 Analysis of the current Conv optimization **7**
3.1 Transformations . . . 7
3.2 Algorithm . . . 8

4 Analysis-based approach to Conv optimization **9**
4.1 Supporting cyclic subgraphs . . . 9
4.2 Supporting additional opcodes . . . 9
4.3 Moving Conv nodes bidirectionally . . . 9
4.4 Removing Conv nodes by increasing bit widths . . . 10
4.5 Indication for a general analysis . . . 10

5 Don't care bit analysis (DCA) **12**
5.1 Prior work . . . 12
5.2 Lattice . . . 12
5.3 Data-flow equations . . . 13
5.4 Transfer function . . . 15
5.4.1 Arithmetic opcodes . . . 15
5.4.2 Information gathering transfers . . . 15
5.4.3 Other transfers of interest . . . 15
5.5 Implementation . . . 16
5.6 Testing . . . 16

6 Conv optimization using DCA result **17**
6.1 Transformations . . . 17
6.1.1 Moving downconv up . . . 17
6.1.2 Contracting downconvs . . . 18
6.1.3 Moving upconv down . . . 19
6.2 Algorithm . . . 19
6.3 Implementation/Testing . . . 20

7 Evaluation **21**
7.1 SPEC performance . . . 21
7.2 Effects on libFirm IR graphs . . . 22

8 Conclusion **23**
8.1 Summary . . . 23
8.2 Future work . . . 23

1 Introduction

The data types of a programming language need to be present in its intermediate representation (IR) as well. In the graph based IR Firm, conversion between types is represented by Conv nodes. Not all of the Conv nodes are actually necessary. Removing the unnecessary ones allows further optimization and no conversion code has to be generated, improving run-time performance.

The libFirm compiler framework already includes a Conv optimization phase, but it is limited to acyclic subgraphs. The current implementation is analyzed in section 3. To overcome its limitations, an analysis-based approach is presented in section 4.

In section 5, a fix-point based analysis is presented to serve as basis for an improved Conv optimization phase, which is presented in section 6. Finally, the performance of the new phase is evaluated in section 7.

Some of the findings during the course of this work were not relevant for the task at hand, but might still promote future work, as suggested in section 8.2.

2 Preliminaries

2.1 Notation

The notation and terminology regarding the Firm IR follows [MTB05] with the following additions.

As mode conversion is discussed elaborately, N, M and W in any formula denote any three modes where $N, M, W \in Int \wedge \text{size}(N) \leq \text{size}(M) \leq \text{size}(W)$. Similarly, N^s, N^u, M^u, M^s, W^s and W^u are defined on the set Int partitioned according to the signedness of types.

Also due to type-heaviness of the discussion, superscripts to opcodes are introduced to denote the mode of its result.

As control dependencies are mostly irrelevant in the conceived optimization, the implicit basic block operand to each op code is omitted from Firm IR syntax and edges in illustrating graphs always denote data dependencies unless noted otherwise.

3 Analysis of the current Conv optimization

Studying the current implementation is expedient for several reasons. The goal of this Studienarbeit is to improve on the current Conv optimization. To avoid regressions in performance, the transformations performed by a new implementation should imply those made by the old one. To avoid regressions in computational complexity, it is essential to study the algorithms used.

Of particular interest is the way the non-orthogonal goals of minimizing bit widths and minimizing the number of Conv nodes are currently traded off. Performing all operations in the minimum required bit-width would probably maximize the number of Conv nodes, and the minimum number of Conv node can probably achieved by performing all computations at maximum bit-width.

3.1 Transformations

The current Conv optimization performs three kinds of transformations: movement of Conv nodes against data flow direction, contraction of adjacent Conv nodes and contracting Conv and Const nodes.

Conv nodes are moved through the graph by transforming a subgraph

$$Conv^N(Op_0^W(Op_1^W, \ldots, Op_n^W))$$

into

$$Op_0^N(Conv^N(Op_1^W), \ldots, Conv^N(Op_n^W))$$

when the following conditions hold:

1. $Op_0 \in \{Minus, Phi, And, Eor, Or, Not, Add, Mul, Sub, Shl\}$, i.e., $Op_0$is contained in a white list of opcodes for which the transformation cannot change program semantics.

2. Op_0^W does not have other users besides the Conv node.

3. A cost function determines that the number of Conv nodes in the IRG won't increase. [1]

Contraction transforms subgraphs $Conv^N(Conv^M(Op^W))$ into $Conv^N(Op^W)$ when $Conv^N$ is the only user of $Conv^W$.

Subgraphs of the the form $Conv^N(Const^W)$ are contracted into $Const^N$ by converting the Const node's target machine value from W to N.

[1] This is how the aforementioned optimization problem regarding the non-orthogonal nature of the Conv optimization is dealt with.

3.2 Algorithm

Informally, the old Conv optimization performs the above transformation using three nested depth-first searches (DFS). The first DFS is performed to locate Conv nodes within the IRG, the second to locate convertible nodes within the data-flow subgraph the Conv node operates in. The third DFS is used to determine the cost of a potential conversion of a node found during the second DFS.

The algorithm outlined above is then run repeatedly until the graph no longer changes, reaching a fixpoint.

While a theoretical worst-case complexity of $O(|E|^3)$ — E being the number of edges in the IRG — implied by three nested depth-first searches sounds expensive for an optimization phase, the run-time of the optimization on IRGs of real-life programs appears small enough for the optimization to not stick out during libFirm profiling.

4 Analysis-based approach to Conv optimization

4.1 Supporting cyclic subgraphs

Figure 1 illustrates a cyclic IR subgraph that could be converted from type W to type N. The classic Conv optimization does not touch this case because it violates condition 2 — the Phi node has multiple users.

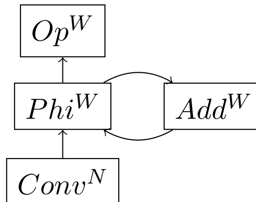


Figure 1: An example of a cycle in the data dependency graph preventing optimization.

At this point, a suggesting, basic approach to support cycles could be to perform a preceding analysis that marks nodes as “safe” if they are only reachable from nodes with compatible, narrow types via paths containing only white listed operations. The only modification to the algorithm would be to check the safe flag instead of the white list - cost function and transformations would continue to operate locally.

4.2 Supporting additional opcodes

Looking at the white list of opcodes more closely reveals that the listed opcodes have in common that higher bits in their operands can never affect lower bits in the operation's result. This property allows to perform the computation with a smaller type if the user of the node only cares about some subset of the bits available in a narrower type.

An example for a white listed opcode is *Shl*. An example of a non-white listed one is *Shr*. Depending on context, however, it can actually be legal to down convert a *Shr* node, too. For example, converting the *Shr* from mode H to mode B is safe in the subgraph $Conv^B(Shl^H(Shr^H(x, 1), 1))$ for any value of x.

Thus, tracking the “caredness” of bits instead of plain connectedness via white listed nodes yields a more precise analysis, allowing optimization of additional opcodes.

4.3 Moving Conv nodes bidirectionally

The old Conv optimization only decreases bit widths by moving Conv nodes to narrower modes (downconv) against data flow direction. While moving Conv nodes to wider modes (upconv) in data flow direction would also decrease bit widths, this is not possible without an analysis-based approach, as an upconv node itself doesn't convey any information on whether the bits that would be lost by moving the node in direction of the data flow are relevant for the program.

In combination with the analysis outlined above, moving upconvs in data flow direction ought to be feasible as long as only not-cared-for bits are lost.

There is another approach to determining the safety of moving an upconv in data flow direction: libFirm includes an analysis that that locates constant bits in data mode nodes (FP-VRP). If the bits that are about to get lost are suitably constant, e.g., all zero with unsigned integer mode nodes, the node can be converted despite cared-for bits being lost because they will be restored later by the upconv with no information loss.

4.4 Removing Conv nodes by increasing bit widths

One of our objectives — minimizing the number of Conv nodes in Firm graphs — could also be achieved by increasing bit widths instead of minimizing them. The platforms libFirm back ends target usually have a native bit width determined by CPU register widths and ALU capabilities. Choosing bit widths below this native width for computations doesn't necessarily improve performance. Even worse, it might degrade performance because alignment constraints of a targeted architecture could require additional code or cause extra bus cycles.

Therefore, choosing a native width instead of narrower ones might result in graphs that contain a smaller number of Conv nodes as well as better performing code.

However, this approach is problematic for two reasons. One, a new interface to firm back ends would be necessary so the Conv optimization could figure out the native type, or — more general — the run time cost of using various types.

The second reason is modulo semantics. The architectures targeted by libFirm back ends exclusively use two's complement integer arithmetic, which implies a modulo operation on each and every computation performed with a modulus depending on bit width of the operation.

Some languages currently translated using libFirm would be consistent with changing this modulus. For example, [KR88] states with respect to using integer types:

"The handling of overflow, divide check, and other exceptions in expression evaluation is not defined by the language."

...which is still true in [ISO99].

However, it was indicated by IPD staff that libFirm IR should guarantee modulo semantics on integer overflow.

4.5 Indication for a general analysis

Early in the design process the following piece of code originating from the "crafty" part of the SPEC CPU2000 benchmark was brought to the author's attention with the comment "it would be nice if the new Conv optimization could do something about this".

```
short f(long long x, long long y) {
    return (x >> 16 | y << 16);
}
```

The resulting libFirm-Subgraph:

$$Conv^{Hs}(Or^{Ls}(Shl^{Ls}(x, 16), Shrs^{Ls}(y, 16)))$$

While the $Conv^{Hs}$ node provides the information that the subgraph starting with Shl^{Ls} is effectively constant, and propagating the *Conv* node would allow following libFirm optimizations to recognize it as such, this would be incompatible with the immediate goal of the Conv optimization, as it would temporarily increase the number of Conv nodes present in the graph. Additionally, the above solution would only work on a special case where non-constant but not-cared-for bits are separated from cared-for but constant bits on the bit width boundary of a mode. A more general solution to optimize these *occult constants* would allow arbitrary mixing of not-care-for bits from our conceived analysis and bits determined constant by the aforementioned FP-VRP.

Including the optimization of occult constants in the Conv optimization does not seem sensible, as both optimizations appear orthogonal apart from the required analysis. It does, however, suggest implementing the outlined care bit analysis as an independent module for use by such orthogonal optimizations.

5 Don't care bit analysis (DCA)

To summarize the findings in the previous sections: In order to overcome the shortcomings of the old Conv optimization, we need an analysis for libFirm that provides information for each data mode node on which of its bits are relevant for a program's computation. We could then continue to tackle the optimization problem by proven means with minor revisions of the algorithm documented in section 3.

5.1 Prior work

The most extensive prior work in the area appears to be [Ste00]. Its author presents a compiler, *Bitwise*, that frees the programmer from declaring bit widths of integer variables by determining necessary bit widths automatically through data flow analysis performed on an SSA IR. In contrast to Firm, the SSA IR used in Bitwise does not represent type conversion itself, freeing the author from the optimization problem of trading the number Conv nodes with minimizing bit widths.

The Bitwise author considered three lattices as candidate for his analysis. One with the abstract values being the minimum number of bits required for a value. The second with bit vectors determining which bits are possibly assigned to. The third is the classical value range analysis lattice.

The Bitwise author chose the latter for maximum precision of the analysis. Experience among IPD staff contraindicated implementing a value-range based bit-width analysis for libFirm, as computing fix points on bit vector lattices usually shows better convergence behavior than range-based ones, and is to be preferred for implementing "yet another" phase of a general-purpose compiler.

Another aspect is that the bitwise compiler performs a bi-directional analysis, combining the classical forward-propagating value range analysis with our conceived backward-propagating care-bit-analysis.

Yours truly was advised to be wary of combining analysis unless it is clear that the combination is more powerful than the constituents in terms of [Cli95]. No evidence of this could be provided in time for this Studienarbeit , so work on design of a uni-directional backward-propagating analysis for care bits was started with the intend to make use of both, the results from the new DCA and the exclusively forward-propagating value range analysis already available in libFirm for performing the actual Conv optimization.

5.2 Lattice

The user of the conceived DCA expects information on each bit in the result of each Firm node about whether it is relevant for the program's computation or not. We define 1 to mean a bit is cared for, i.e. relevant, and 0 to mean a bit is not cared for, i.e. irrelevant.

A typical textbook lattice, as well as the bit vector lattice considered by the bitwise author, contains additional $\bot$ and $\top$ elements to denote "not enough" and "contradicting" information, as illustrated on the left of figure 2. However, we can use the simpler,

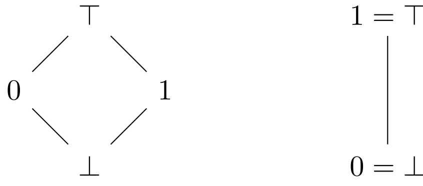


Figure 2: Hasse diagrams of two lattices considered for bits of the don't care analysis. On the left a standard textbook one. The right one was chosen for this analysis.

two-element lattice on the right for the following reasons.

1. There is no way one could imagine a "contradiction" with our analysis at hand, so this state would be unused.

2. Discerning the states "no information" and "cared for" wouldn't be of any value to the conceived users of the DCA, so these states can be coalesced into "cared for" with respect to the users.

3. Coalescing "no information" and "cared for" is also ok for analysis purposes if we allow for the fact that the abstract values are not conservative approximations until the algorithm for computing the fixpoint terminated. If, upon termination, an abstract value was to be stuck on its initialization value "not cared for" because it wasn't reached, the approximation is correct, as the variable is part of unreachable code, implying irrelevance of its constituent bits.

The advantage of the four-element lattice appears irrelevant when compared with simpler data structures and more concise code possible with computation on a two-element lattice.

Now that we have fixed the bit-wise lattice as $L = \{1, 0\}$ with boolean $\vee$ and $\wedge$ as infimum and supremum operators, we define the lattice for bit vectors as $V = L^w$ whereas w denotes the width of the widest possible mode available in libFirm, and the infimum and supremum operators applied component-wise.[2]

To cut down on notational overhead when defining the transfer function, bit vectors will be noted as binary numbers with bit-wise application of the operators. Figure 3 illustrates the resulting product lattice in this notation for $w = 3$.

5.3 Data-flow equations

Given a set of bits the user cares about in the result of a libFirm opcode, we can determine which bits in the operands can affect it. We formalise this as the *cares*

[2]In the implementation, the bit width of the abstract value vectors actually varies according to the width of the mode of the node the value is associated with to make the result of the DCA consistent with previously implemented analyses in libFirm. This fact is dismissed in this theoretical presentation as yet another implementation detail.

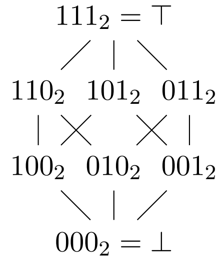


Figure 3: Hasse diagram of the product lattice used for a hypothetical DCA on variables with a width of 3 bits.

function:

$$\text{cares} : O \times V \times \{1, \ldots, \text{arity}(O)\} \longrightarrow V$$

O being the set of firm opcodes, V our lattice, and the third argument selects the position of the operand we are interested in.

So far, the discussion only defined abstract values for data mode nodes, as this is what the intended users of the DCA require. For analysis purposes, we have to extend the semantics for the rest of the libFirm modes. In case of nodes with Memory, Execution or Basic Block modes, the lattice semantics degenerates into an unqualified "care"/"don't care" indicator represented by the $\top$ and $\bot$ elements, turning the DCA into a plain reachability analysis for nodes of these modes.

To illustrate the interaction between different modes during analysis: If a user would care for certain bits in a Phi node, the caredness information does not solely propagate to the Phi node's data operands. The value of the bits cared for in the result of the Phi node can be mutated by choice of control flow to reach the Phi node, so we also care for the Phi's BB operands. Following control dependencies might in turn lead to unqualified "caring" for a Cmp node, again leading to caring of data mode bits in the operands of the Cmp node.

With all Firm nodes and modes covered, we can now formalise the following data-flow equations on IRG nodes N:

$$\text{in}(N) = \bigvee_{s \in \text{succ}(N)} \text{out}(s, \text{position}(N, s)) \tag{1}$$

$$\text{out}(N, n) = \text{cares}(\text{opcode}(N), \text{in}(N), n) \tag{2}$$

$$\text{out}(End, n) = \top \; \forall n \in \{1, ..., arity(End)\} \tag{3}$$

If our *cares* function is designed to be monotone with respect to its lattice parameter, we can now determine caredness in bits of nodes by computing the minimum fixpoint.

5.4 Transfer function

5.4.1 Arithmetic opcodes

Propagation of the care bits in arithmetic modes directly follows from the definition of the respective operations and the definition of our care bits. Only a selected subset of the definition of the *cares* function is presented here.

- The *Add* opcode cares for all bits in its operands up to the highest bit present in the caring user. For example, $cares(Add, 1000_2, 0) = 1111_2$. This is our best approximation, as a carry operation caused by any of the lower bits might affect the bit the user cares about.

- Bit wise, symmetric operations, such as *And*, *Eor*, hand the abstract value right through.

- If the shift count of *Shl* or *Shr* is constant, they perform the inverse operation on the abstract value, e.g., $cares(Shr(x, 1), 1000_2, 0) = 10000_2$. If the shift count is not constant, caring for $\top$ in the first operand is our only choice. In any case, the second operand is always cared for unqualifiedly ($\top$).

5.4.2 Information gathering transfers

The above definition of the *Shl* transfer could be considered an information gathering transfer, as — in contrast to the previous ones — it can actually introduce "not cared for" bits in its operands, despite the result being cared for unqualifiedly.

The following subset of the *cares* definition shows further gathering transfers.

- Conv nodes, in the case of a down conversion, mark the bits lost during the conversion as "not cared for". E.g., $cares(Conv^B, \top, 1) = 11111111_2$.

- If arithmetic opcodes have suitable constant arguments, they can yield not cared for bits in their operands. For example, if an *And* node has a Const node as one operand, the bits cared for in the other operand are determined by the incoming abstract value masked with the constant of the Const node.

5.4.3 Other transfers of interest

- Conv nodes, in the case of an upconv, do yield a nontrivial definition when the mode of the operand is a signed integer one using two's complement arithmetic due to sign extension, and the user of the Conv node is interested in one of the sign extended bits. Assuming the operand of the Conv node in the following map is of mode *Bs*, the transfer could look like $cares(Conv^H, (1, 0000, 0000_2), 1) = (1000, 0000_2)$.

5.5 Implementation

The DCA was implemented as a separate module in libFirm to facilitate use by other optimizations besides the Conv optimization, for example an optimization for the occult constants described earlier.

The fixpoint itself is computed by means of the Work list algorithm.

Liberal use of assertions was made in order to protect against an accidental non-monotone transfer during revisions of the code.

The analysis is available as

```
void dca_analyze(ir_graph *irg);
```

Upon computation of the fixpoint, the result is available in nodes in the form of links to target values containing the care bits.

5.6 Testing

Testing the DCA code and conservativeness of the fixpoint indirectly via the Conv optimization is not effective as only a minute part of the analysis actually results in transformations of the graph. To thoroughly test the DCA, a fuzz testing approach was used by implementing a graph walker that inserts Eor nodes to toggle bits that have been analyzed as "don't care".

When fuzz-testing was found to change the semantics of a program during development, and the reason was not obvious, an efficient way to systematically locate the bug was restricting the fuzz-walker to certain node IDs modulo 2^n. A binary search would then quickly locate the bogus abstract value.

6 Conv optimization using DCA result

To use the analysis result for optimization, two problems need to be solved. First, legal transformations need to be found. The fact that only not-cared-for or constant bits would be lost on conversion of a node from mode W to mode N does not imply that it is safe.

Some of the reasons are implementation details hushed up in most publications about libFIRM, such as the "modulo shift" problem, where the implicit modulus applied to the shift count argument of a shift operation does correlate with the mode of the shift node's result in non-intuitive ways.

Others stem from the fact that the old Conv optimization liberally switches signedness. The DCA doesn't make any predication about whether switching signedness of modes is safe, but as we set out for a no-regressions solution, we can't dismiss this feature.

Yet another source of concern are nodes whose nature is not entirely reflected in the IR graph, thwarting a more elegant analysis by requiring special cases. An example for such a node is the Mulh node, which could be considered a graph macro expanding to the subgraph in figure 4.

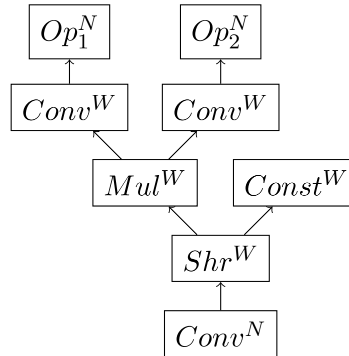


Figure 4: An IR subgraph equivalent to the Mulh node.

The second problem is determining whether a transformation is favorable. Simply modifying the graph to perform every operation in the smallest mode possible would maximize the number of Conv nodes, which would be contrary to the goal of the classic Conv optimization.

6.1 Transformations

The new Conv optimization recursively performs three transformations on the IRG .

6.1.1 Moving downconv up

The transformation in Figure 5 is applied when all of the following conditions hold:

1. The highest cared for bit in Op_1^W does fit into mode N.

2. Op_1^W doesn't care for bits in its operands higher than $size(N) - 1$

3. Op is an opcode that won't change program semantics on down conversion under condition (2).

4. A cost function determines that the number of Conv nodes in the IRG won't increase.

For most opcodes, condition (2) implies condition (3). For some opcodes however, such as the aforementioned Mulh node, this is never true if a user cares about any of the bits in its result.

Conditions (1) and (2) also guarantee that the fixpoint doesn't change, and remains a conservative approximation.

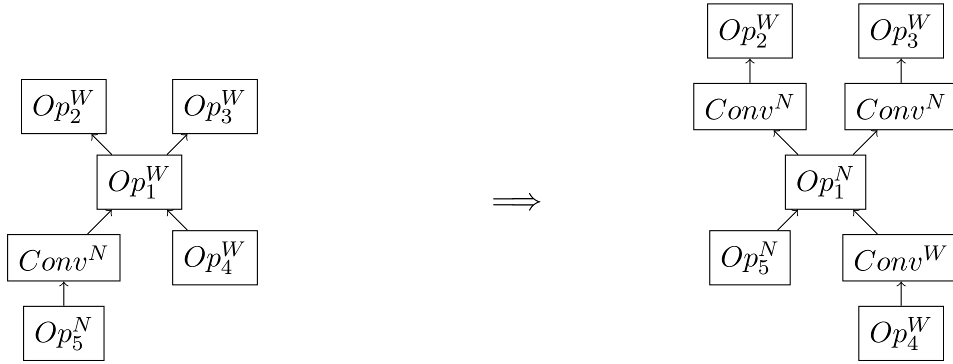


Figure 5: Converting Op_1^W by moving $Conv^N$ up.

6.1.2 Contracting downconvs

As with the classic Conv optimization, contraction transforms subgraphs

$$Conv^N(Conv^M(Op^W))$$

into

$$Conv^N(Op^W)$$

when $Conv^N$ is the only user of $Conv^W$.

This restriction is not limiting the new Conv optimization, as we can mediately contract Conv nodes with multiple users by applying the Transformation in 6.1.1 first.

This transformation also doesn't change the DCA fixpoint, because

$$\text{cares}(Conv^M, \text{cares}(Conv^N, \top, 0), 0) = \text{cares}(Conv^N, \top, 0)$$

6.1.3 Moving upconv down

In order to move upconvs in direction of the data flow, the Transformation in figure 6 is performed when the following conditions hold:

1. The highest cared for bit in Op_1^W does fit into mode N
2. Op_1^W doesn't care for bits in its operands higher than $size(N) - 1$
3. Op is an opcode that won't change program semantics on down conversion under condition (2).
4. A cost function determines that the number of Conv nodes in the IRG won't increase.

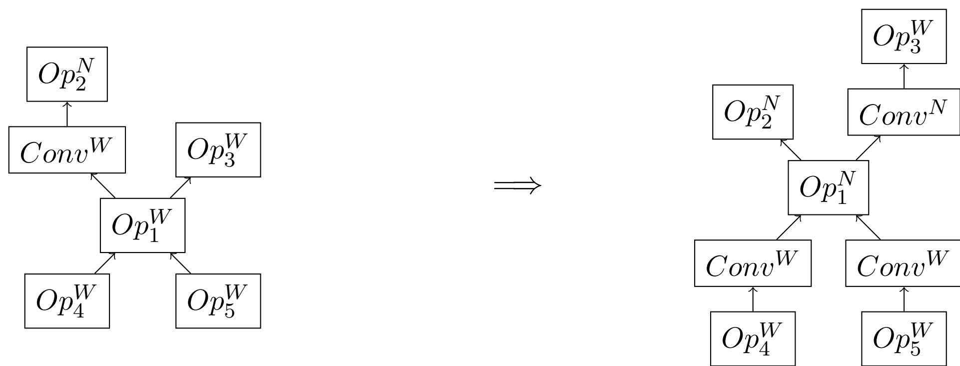


Figure 6: Converting Op_1^W by moving $Conv^W$ down.

6.2 Algorithm

As worked out in section 3, the old Conv optimization can be made analysis-driven by consulting the result of the DCA for local decisions. Naturally, the algorithm for type conversion by moving Conv nodes around can be mostly reused.

In order to allow for moving upconvs in direction of the data flow, the depth-first searches now also follow edges in that direction, and the procedures for modifying the graph as well as the cost function have been adapted.

However, there is one major change to the algorithm. Due to the added support for cycles in the dependency graph, and the continued support to liberally switch signedness around, we can no longer call the nested depth-first searches repeatedly until a fixpoint is reached, as there might always be a Conv node that can be moved around to convert a path to its "smaller" unsigned/signed type.

The easy way out of enforcing antisymmetry on the type conversion relation would violate our goal of a no-regression solution. The problem was "solved" by performing the inner algorithm only once instead of repeatedly. While this also means accepting a regression, testing has shown that enforcing antisymmetry on the type relation yields a much more severe penalty than omitting repeated computation.

6.3 Implementation/Testing

Implementation was done by revising the old Conv optimization.

To allow additional coverage during testing, a “stress test” mode was added. If activated, the cost function always returns “favorable”. This was deemed necessary because the theoretical discussion did not make any assumptions about the correctness of the cost function, which should be reflected in the code.

Test	run time old [s]	run time new [s]	change [%]
gzip	108.01219	108.90952	0.8
vpr	87.572122	89.003499	1.6
gcc	52.607665	52.296054	-0.6
mcf	55.950885	55.740288	-0.4
crafty	52.317615	52.683097	0.7
parser	119.79726	122.79636	2.5
perlbmk	92.652055	91.468321	-1.3
gap	54.163619	55.019924	1.6
vortex	115.87767	98.742977	-14.8
bzip2	85.1815	84.693048	-0.6
twolf	124.97449	125.44915	0.4
mesa	118.38633	119.72873	1.1
art	64.373353	64.662043	0.4
equake	75.954529	75.816939	-0.2
ammp	238.16081	241.16126	1.3
Total	1445.9821	1438.1712	-0.5

Table 1: Comparison of run time of benchmark tests between old and new Conv optimization.

7 Evaluation

In order to evaluate the new Conv optimization, we will compare it to the old version by compiling and executing the SPEC CPU2000 benchmark, available from [Sta], with each version respectively, observing changes in the runtime of the produced programs.

The exact libFirm revision used for the comparison is 1.21.0. The cparser revision used was git revision f0a1b89b. The Conv optimization version was taken from the author's repository [3] and transplanted on top of the versions above, as the current development tree of libfirm would not compile some of the SPEC test programs.

7.1 SPEC performance

The values for comparison were obtained by performing the benchmark eight times for each version and computing the minimum value for each test in order to reduce uncertainties introduced due to the test systems available to the author being networked multi-user multi-tasking machines. The machine itself is not detailed further, as we are interested in relative changes only.

Table 1 lists the runtime values.

[3] The exact revision being 7a2c079a9d3011bb2036ef75fce13cf13efee7b6.

Test	ΔConv old	ΔConv new	Δbits old	Δbits new
ammp	-5	-14	-128	-1160
equake	0	0	32	-192
art	0	0	0	-64
mesa	-813	-772	-7304	13592
parser	-73	-122	-384	-20064
gcc	-1384	-1312	-9136	-50016
vpr	-29	-13	80	-288
mcf	0	0	32	-64
vortex	-98	-106	-888	-5928
gzip	-131	-141	-2696	-3000
crafty	-353	-354	-9496	-11696
twolf	-299	-296	-1840	-9808
gap	-1108	-1156	-18800	-13208
perlbmk	-811	-1352	-11328	-29736
bzip2	-39	-34	-880	-808
total	-5143	-5672	-62736	-132440

Table 2: Changes made to IRGs of tests of the SPEC in terms of Conv nodes removed and bits minimized.

7.2 Effects on libFirm IR graphs

Beyond comparing the actual benchmark results it is also interesting to compare the changes made to firm graphs by both versions.

In order to condense the changes to tangible numbers, both, the old and the new Conv optimization were augmented by an evaluation graph walker that counts the number of Conv nodes in IRGs before and after an optimization run. The counts before the run are subtracted from the ones after the run and in turn aggregated by summing over multiple runs of the optimization on each IRG, over multiple IRGs inside an compilation unit, and over multiple compilation units constituting the respective SPEC tests.

In order to compare changes related to the bit-width minimization part of the Conv optimization, the evaluation walker also counts the total number of bits in integer mode nodes other than Conv nodes. The numbers are then aggregated the same way as above. For example, if during the evaluation run the mode of a single non-Conv node would have been changed from LL to L, the result would be -64.

The results are shown in Table 2. In summary over the entire SPEC code, the new Conv optimization shows a 10% improvement regarding the number of Conv nodes removed from IRGs and obliterated twice as many bits in non-Conv nodes of IRGs than the old one.

Individual tests do show regressions in the number of Conv nodes or bits removed. A possible explanation could be the shortcomings mentioned in 6.2. A review of the concrete IRGs involved might bring clarity, but sadly the author had no time left before deadlines to investigate further...

Test	occult constants
gcc	24
gap	14
bzip2	10
others	0
total	48

Table 3: Occult constants found in IRGs during compilation.

8 Conclusion

8.1 Summary

In this Studienarbeit, we designed and implemented an improved version of the the Conv optimization in libFirm using data flow analysis techniques. As the analysis that was developed in order to reach this goal appears to have additional value besides its use for Conv optimization, it was implemented as a separate module to allow further application.

8.2 Future work

As previously suggested, the DCA result is valuable for optimizations beyond bit width minimization.

By combining DCA and FP-VRP bit vectors, occult constants can be located and replaced by real constants, in turn allowing further constant propagation and constant folding. Searching for occult constants within IRGs of the SPEC revealed that only three of the test programs contain occult constants, as detailed in table 3. The fact that such an optimization comes at negligible cost when FP-VRP and DCA analyses were already run might make an implementation attractive despite the low number of occurrences.

It might also be worthwhile to investigate if combining FP-VRP and DCA, both unidirectional analyses, into a bidirectional fixpoint analysis yields a more powerful analysis in terms of [Cli95].

References

[Cli95] Clifford Noel Click, Jr. *Combining analyses, combining optimizations.* PhD thesis, Houston, TX, USA, 1995. UMI Order No. GAX96-10626.

[ISO99] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall Software Series. Prentice Hall, 1988.

[MTB05] Götz Lindenmaier Martin Trapp and Boris Boesler. *Documentation of the Intermediate Representation Firm, the Firm Intermediate Representation Mesh.* Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, 1.2 edition, 2005.

[Sta] Standard Performance Evaluation Corporation. CPU2000 benchmark. http://www.spec.org/cpu2000/.

[Ste00] Mark William Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Master's thesis, Massachusetts Institute of Technology, 2000.