Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Verbesserte Optimierung von Integer-Konvertierungen und VHDL-Codeerzeugung mittels Bitbreitenanalyse

Bachelorarbeit von

Marcel Hollerbach

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Andreas Fried
Bearbeitungszeit:	1. Januar 1990 – 31. Dezember 2000

Zusammenfassung

Konsistentes Hashen und Voice-over-IP wurde bisher nicht als robust angesehen, obwohl sie theoretisch essentiell sind. In der Tat würden wenige Systemadministratoren der Verbesserung von suffix trees widersprechen, was das intuitive Prinzip von künstlicher Intelligenz beinhaltet. Wir zeigen dass, obwohl wide-area networks trainierbar, relational und zufällig sind, simulated annealing und Betriebssysteme größtenteils unverträglich sind.

Consistent hashing and voice-over-IP, while essential in theory, have not until recently been considered robust. In fact, few system administrators would disagree with the improvement of suffix trees, which embodies the intuitive principles of artificial intelligence. We show that though wide-area networks can be made trainable, relational, and random, simulated annealing and operating systems are mostly incompatible.

Ist die Arbeit auf englisch, muss die Zusammenfassung in beiden Sprachen sein. Ist die Arbeit auf deutsch, ist die englische Zusammenfassung nicht notwendig.

Das Titelbild ist von http://www.flickr.com/photos/x-ray_delta_one/4665389330/ und muss durch ein zum Thema passendes Motiv ausgetauscht werden.

Contents

1. Introduction **7**

2. Basics **9**
2.1. cparser / libfirm 9
2.1.1. Number representation 10
2.2. Lattice 10
2.3. Fixed point iteration 11

3. Design & Implementation **13**
3.1. Bitwidth analysis 13
3.1.1. Value prediction 14
3.2. Stable Conversion nodes 17
3.3. VHDL generation 18
3.4. How about section 19
3.4.1. Value range vs. bitwidth 19
3.4.2. Widening & Narrowing 20

4. Evaluation **21**
4.1. General runtime 21
4.2. Usage in vhdl generation 21

5. Conclusion **23**
5.1. x86 generation & vhdl generation 23
5.2. Further improvements 23
5.3. Additional analyzer usage 23

A. Sonstiges **29**
A.1. Table of node rules 29

1. Introduction

The problem to solve is called bitwidth analysis. The bitwidth with a data word can be seen as the bits that are actually used in the runtime of the source code. Lets take the following example:

```
int arr[4];

for (int i = 0; i < 4; i++) {
  int res = (i << 4) + i*i;
  arr[i] = res;
}
```

After running the bitwidth analysis, every operation has a attached information structure which indicates how many bits are actually used by the code. The developed algorithm applied to the the code results in something like this: $i \in 0..3$; $res \in 0..90$;

2. Basics

2.1. cparser / libfirm

Modern compilers are now developed for half a century. Over the time a new structure has evolved. Compilers are split into two parts. The first part is called *front end* and handles everything related to the language specific parts. The second part is called *back end* and handles the translation into something like assembler or java bytecode. As an example, cparser is handling the c specific tasks. For the generation of the hardware instructions *lib*Firm is used.

Control flow graph in libFirm The interaction between the front and back end is based on a data structure called control flow graph (CFG). A CFG is a directed graph. The relation from a node A to a node B can be interpreted as node A having node B as a operand. CFGs are also the base structure for analyzing the control flow of a software. Such a analysis is called control flow analysis.

In *lib*Firm the nodes of the CFG can have different types: control flow, arithmetical operations, memory handing, constant expressing. A node is additionally placed in a block. A node is executed if all its operands have been executed. Or, if there are no operands, when its block is executed. Another software that does the same thing is *clang*, where the back end is handled by llvm. A more in detail explanation can be found in TODO TODO.

interaction from front to back end As mentioned earlier, *lib*Firm users are building up a CFG using libFirms API. Now let us take cparser to see a example how the front end and the backend are working together. We want to compile the programcode from 2.1.
First of all the front end will parse the given source code. The parsed code then will be stuffed into a abstract syntax tree (AST). The representation in the abstract syntax tree will encode the syntactical informations from the software. Informations like curly brackets will not be needed anymore.
After that the AST will be transformed into a CFG.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char const *argv[]) {
  printf("Hello Firm!");
  return 0;
}
```

Figure 2.1.: Sample source code

value	7	6	5	4	3	2	1	0
5	0	0	0	0	0	1	0	1
-2	1	1	1	1	1	1	1	0

Figure 2.2.: Number representation

At this point the front end handed the informations for binary creation to the back end. However, the front end can now tell the back end to perform optimizations on the information. In terms of C this is often controlled with the -O1,2,3 flags.
At the last stage the back end is told to output a binary.

2.1.1. Number representation

So called *modes* are used in libfirm to represent data words where operations are performed on. Such a mode has a length. A mode also has a type, the interesting types for us are the number style types. Which are integer and float. The most interesting for us are the integer-mode, since these are the types where we are performing our analysis on.

Those integer-modes can be signed or unsigned. The length of the mode defines the maximum number. If it is signed, then it also defines the minimum number.Otherwise the minimum is just null. The sign is encoded using two's complement. As an example, in Figure 2.2 two numbers are displayed with its hardware representation.

The term *mode* here is used to describe integer-modes for the rest of this text.

2.2. Lattice

A lattice is a algebra structure. There are the following rules for the them:

- Finite number of elements
- $\vee : V \times V \to V$ will return the smallest element, that is bigger or equal than the two operands
- $\wedge : V \times V \to V$ will return the biggest element, that is smaller or equal than the two operands
- $\forall a, b \in V : \exists a \vee b$ AND $a \wedge b$
- $\forall u, v \in V : u \vee (u \wedge v) = v$ AND $u \wedge (u \vee v) = v$

The element, which is the smallest from all elements, is called $\bot$. The biggest of all is called $\top$.

A lattice can also be made visible in a Hesse diagram, which you can see in Figure 3.1

2.3. Fixed point iteration

Fixed point iteration is a method for finding a fixed point, while iterating over a function.
The function we use for iterating is called f, and is defined as $f : D \to D$. F is continues. We call $x \in D$ a fixed point if $f(x) = x$.

fixed points in compiler analysis We can transform and use the fixed point iteration to be working on a graph G. Each node has a number and a element from the lattice. In the beginning the element is $\bot$. We call the number of the node *id*. The lattice element is called *v*. The set D consists of elements of the form $\{id, v\}$. Where we can say that $|D| = |G|$, and every node from G is part of D. The function f then calculates for every node, if a better v can be found. If there is one, then a new tuple $\{id, v\}$ with the better value will replace the old tuple in D. If we now hit the situation where $f(x) = x$, then we know that we have found a fixed point, which is the best correct solution we could find.

3. Design & Implementation

3.1. Bitwidth analysis

The analysis is a data flow analysis. The analysis attaches a $(int, boolean)$ tuple to every meaningful node. A node is considered meaningful is the node has a integer-mode. We will reference the first value as *stable bits* and the second bit as *is positive*. What stable here means can be observed while looking at 2.2. For the first line of the table we have five stable digits. The second line has 7 stable digits.
Since this is a data flow analysis that is build on a iterative approach we might want to address the current tuple of a node x, while we have a new one calculated. Therefore we call $\hat{x}$ the currently associated value, and x the newly calculated one.

Bit representation The stable bits are indicating how many bits are stable, and therefore not used. The second value of the tuple indicates if the value will ever reach negative numbers or not. And thus indicate at least one stable bit at the highest position. However, the second value is only meaningful for modes that allow signs.

Range representation There is also a second way of interpreting the two values. The stable bits can define a minimum and maximum range. The maximum number is reached if the stable bits are just always zero. If the mode is signed and the node is not positive, then the minimum number is reached by assuming all stable bits are one. Otherwise the minimum range is 0. We can define the following min max definitions for the ranges:

$$max_{bitwidth}(x) = \begin{cases} 2^{stable\_digits-1} - 1 & mode.signed \\ 2^{stable\_digits} - 1 & Otherwise \end{cases}$$

$$min_{bitwidth}(x) = \begin{cases} 2^{stable\_digits-1} & mode.signedandis_positive \\ 0 & Otherwise \end{cases}$$

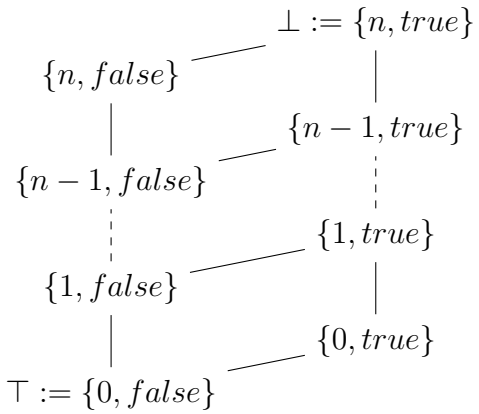

Figure 3.1.: The definition of a upper bound compare node

Analysis The analysis works as a fixed point iteration. Therefore we use Figure 3.1 as lattice. The values of the lattice are representing the tuples from the analysis.

As a first step, we iterate over every single node and initialize the node with $\top$ and mark it as *dirty*. If the node is constant, we calculate its bitwidth. Nodes with the opcodes *Const*, *Size* and *Address* are considered constant.

The second step consists of recalculating every *dirty* node in the graph. if the stable bits of x are greater than those from $\hat{x}$, then the new value is memorized as $\hat{x}$ of the node and every successor of the node is marked as dirty. The used rules for recalculating the nodes are described in section A.1.

3.1.1. Value prediction

In addition to the normal analysis results, the fixed point iteration can insert additional confirm nodes. Those confirm nodes help making the analysis more accurate. First of all we need a few definitions for easier understanding:

Definition: True / false Blocks A compare node in libFIRMhas always a relation, 2 operands and 2 blocks. One of the blocks is executed depending on if the criteria was meet or not. The block that is executed when it was meet, is called *true-block* the other block is called *false-block*. The structure is visualized in Figure 3.2

Definition: Upper bounds A node defines a upper bound if the relation is $<$ and the second operand is constant.

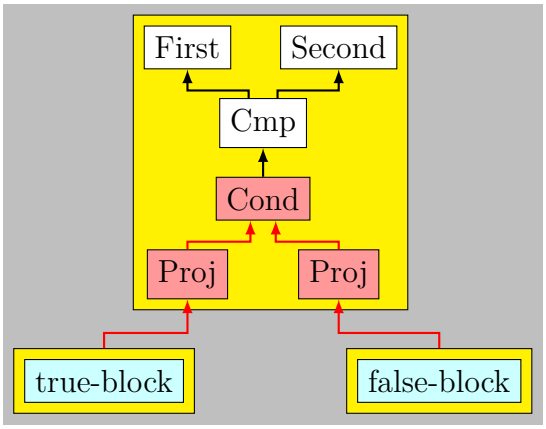

Figure 3.2.: The definition of a upper bound compare node

A compare node is also defining a upper bound if it can be transformed into a construct that matches the definition. For example by switching the right and left nodes, while turning the relation.

Definition: Predecessor in a certain block In the detection described later, we often need to find a predecessor that is placed in a certain block. Therefore we define:

$$\kappa(a, b) := \{X | X \leftarrow a \wedge X.block = b\}$$

It will return every node that is located in b and is a predecessor of a.

Definition: Constant dependencies While looking at C code we often see that addition and multiplication nodes are used for calculating array addresses or addresses for structure access. Therefore one operant of the arithmetical operations is often constant. A example for this can be found at Figure 3.3. We define ξ to explore the whole tree of one constant one not constant operants, and return us every node that was not constant.

$$\xi(a) := \begin{cases} a \cup \xi(c) & \text{, If there is only one not constant dependency } c \\ \emptyset & \text{, otherwise.} \end{cases}$$

If a has only one not constant operant c, then ξ returns the element a and $\xi(c)$. Otherwise it returns a empty set. For the example the highlighted nodes will be part of $\xi(Y)$.

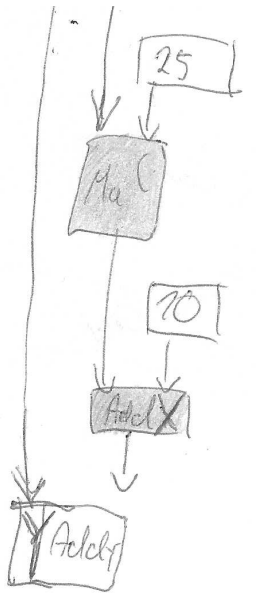

Figure 3.3.: Definition of xi

Upper bounds for block execution The values that are calculated in a node are (even if the fixed point iteration is not stable yet) possible values. The iteration starts at $\bot$ and moves into the direction of $\top$. We now evaluate a upper bound compare node, every time the first operand changes. In the beginning we can say that that with those possible values, each time the true-block will be executed. However, if $max_{bitwidth}(n)$ grows enough to get bigger than the second operand or the compare node, then we can say that we have found a upper bound for the execution of the true-block. Which is $max_{bitwidth}(n)$ in the current state. Thus we can insert a confirm node between every node $e \in \kappa(i, j)$ and i. Where i is the first operand and j is the true block.

Extended confirm insertion The confirm nodes that we have inserted in the paragraph before can also be transported backwards. With $\xi(i)$ we can get a set of nodes, where the current state in the analysis is only depending on one node. Thus we can say that the state of every node from $\xi(i)$ will not change as long as the most upper element in the tree structure does not change. Which means that we can for every $e \in \xi(i)$ and $g \in \kappa(e, j)$ insert a confirm node between (e, g)

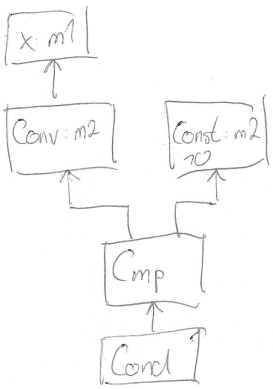

Figure 3.4.: Conversion compare construction

3.2. Stable Conversion nodes

In libfirm a conversion node can be used to convert one dataword from one mode to another. This type of node has one operand. Such a conversion has two effects. The bitwise representation stays the same. Or the bitwise representation changes, due to the numerical representation changes. We call the first case *Stable Conversion node*. A example for a unstable conversion node may look like $(unsigned)((int) - 10)$. A stable conversion node might look like $(unsigned)(int)10$.

Finding stable conversion nodes Such stable conversion nodes can be found using the bitwidth analysis. We compare the range from the operand with the range from the conversion node itself. If the ranges are the same, then we know that the conversion is stable.

Removing conversion nodes In case we found a stable conversion node, then we can say that this node only exists for syntax rules, there is no semantical value in them. Removing those nodes also has the advantage of helping other analyses. The confirm insertion algorithm of libfirm is searching for assertions that can be made based on looking at compare nodes. This works quite well. However, the example at Figure 3.4 does not work The insertion code could only insert a Confirm between the Compare and the conversion. However, it would need to be behind the conversion node, in order to help the rest of the code, since other nodes will likely depend on the operand of the conversion node and not on the conversion node itself. After removing the conversion node, the analysis can find a assertion based on the compare node. This also helps the branch prediction and dead code elimination.

However, for really removing the conversion nodes, we need to find situations where

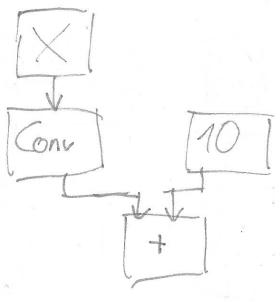

Figure 3.5.: Arithmetical optimization example

we can eliminate the conversion node. We have already seen the example with a compare node. Additionally we can do the same with a arithmetical operation.

Compare-Conversion optimization There is the rule in libfirm that the two operands of a compare node are forced to have the same mode. This means, when we remove the conversion node, we also need to adjust the mode of the second operand. This is not easily possible for something that is not a constant, thus we confine for now that the two operands need to be a conversion and a constant node. With the assertion that our second operand is a constant, we can simply adjust its mode. And remove the conversion.

Arithmetical-Conversion optimization A other situation where we can optimize the conversion nodes are arithmetical operations, where one operand is again constant. In this case we don't really remove the conversion node, we more drag it through the arithmetical operation, and place it afterwards. We do this and hope that we can remove it with a compare conversion optimization after that. Such a construct might look like Figure 3.5 . After we adjusted the mode of the constant we can move the conversion from the operand of the arithmetical operation to the end.

3.3. VHDL generation

Projects like libva have made the first step into the direction of hardware accelated video decoding on a computer system. The projects tries to provide standardized access to the for decoding video streams like MPEG-2, H.264/AVC, H.265/HEVC, VP8, VP9, VC-1. The project itself handles the passing and messaging from a front end application like a video player, to the driver itself. The driver itself then answers the calls from the API. And the decoded picture goes back to the front end application. However, the driver itself still needs to do the decoding on its

side. Preferable implemented in hardware, since this is usually providing the best performance. The amount of work needed for implementing such a decoder for MPEG-2 can be found in FIXME. While looking at this it seems like a good idea to think about writing the code in a higher level language and let it simply compile against vhdl. Exactly this is done by the firm tool *firm2vhdl*.

firm2vhdl The tool provides the transformation from a libfirm graph into vhdl code. Every node in the firm graph gets transformed into a instruction in vhdl. This is done in done by calculating the effect the node has in vhdl code, by using its operands, assigning the result to a new variable. Which then can be used again later by the next operation. Each of those those variables are represented in hardware, and thus have a amount of bits. Which is, at this state, the number of bits of the mode.

bitwidth in firm2vhdl Using the number of bits for a bit can be very wasteful in vhdl as this is wasting a lot of space on the FPGA chip later on. Minimizing the amount of bits used per variable here can be important, since most of the variables used in c are not using the complete bandwidth.
The bitwidth information gathered from the analysis described before can help here, as it defines how many bits of a node are used, and how many are not. Thus we can add code to the transformation, for taking the used bits, instead of the number of bits in a mode.

3.4. How about section

This section is about decisions that where made, and why they are made.

3.4.1. Value range vs. bitwidth

The analysis previously explained is outputting bitwidth informations. The fixpoint iteration uses the set of rules described in section A.1. The same analysis could use a set of rules where we don't calculate the bitwidth, but rather the direct range of possible values (Not bound to 2^x form). There is already such a implementation in libFirm, its called value range propagation (VRP).

However, there should be some way to compare both models. We know that every node in libfirm has a mode, this mode is defining the length of a data word. The

analysis is measuring how many of the bits from this data word are used. So once a node gets marked dirty, there is some sort of delta, like $\delta = max_{bitwidth}(x) - max_{bitwidth}(\hat{x})$. In case of bitwidth, we know that the delta will have the form of 2^x. In case of the value range detection, the only assertion we can have is $\delta = 1$. Now lets take a graph like ???, where X is some sub graph that performs a operation. Under the assertion that the delta of last node is minimal, then the bitwidth analysis is faster. A real life example where this is happening is for example the simple access to a array, like $array[i]$.

3.4.2. Widening & Narrowing

4. Evaluation

4.1. General runtime

4.2. Usage in vhdl generation

5. Conclusion

5.1. x86 generation & vhdl generation

5.2. Further improvements

5.3. Additional analyzer usage

Bibliography

Erklärung

Hiermit erkläre ich, Marcel Hollerbach, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Sonstiges

A.1. Table of node rules