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.

Inhaltsverzeichnis

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

2.1.1. Number representation

2.2. Lattice

2.3. Fixed point iteration

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 style mode. We will reference the first value as *stable bits* and the second bit as *is positive*. 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:

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

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

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

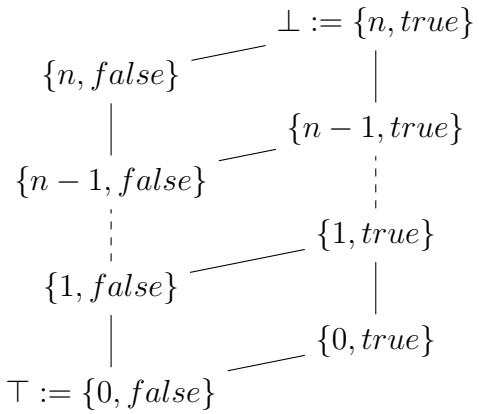

Abbildung 3.1.: The definition of a upper bound compare node

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 REFERENCE TO TABLE

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 3.2

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

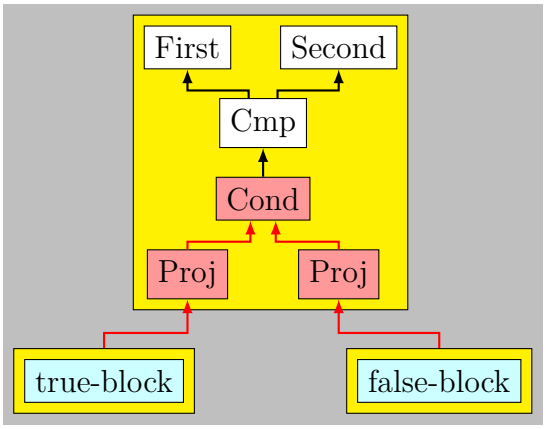

Abbildung 3.2.: The definition of a upper bound compare node

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. Knowing this, a structure like FIXME can be created. 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.

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 $\perp$ 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)

3.1.2. Difference to VRP

There is already a analysis that is doing something similar, it is called value range propagation. The difference from VRP to BA is that in VRP each iteration is trying to predict the exact range after each operation. While BA tries to predict the unused bits after each operation. This little detail is mainly showing up in speed of the fixed point iteration, VRP converges way slower than BA. Details for this are given in the evaluation chapter 4.

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 following construction 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 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

4. Evaluation

4.1. General runtime

4.2. Usage in vhdl generation

4.3. Improvements over VRP

4.4. Widening & narrowing

5. Conclusion

5.1. x86 generation & vhdl generation

5.2. Further improvements

5.3. Additional analyzer usage

Literaturverzeichnis

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

Danke

Ich danke meinen Eltern, meinem Hund und sonst niemandem.

A. Sonstiges

A.1. Anmeldung

Üblicherweise melden wir eine Arbeit erst an, wenn der Student mit dem Schreiben begonnen hat, also nach der Implementierung. Das verringert die Bürokratie und den Stress, der mit verpassten Deadlines kommt.

Außerdem ist ein Abbrechen nach der Anmeldung ein offizieller Akt für den es wiederum Fristen gibt:

	Abbruchfrist nach Anmeldung
Bachelor	4 Wochen
Master	2 Monate
Diplom	3 Monate

Nach dieser Frist muss die abgebrochene Arbeit mit 5,0 bewertet werden.

Das ISS empfiehlt, dass Studenten sich zusätzlich selbst im Studienportal anmelden. Das könnte die Eintragung der Note beschleunigen.

A.2. Antrittsvortrag

Bei internen Arbeiten jeglicher Art ist ein Antrittsvortrag optional. Bei externen Arbeiten ist ein Antrittsvortrag Pflicht.

Dauer: 15 Minuten + 5 Minuten Fragezeit.

Ein Antrittsvortrag sollte nach der Einarbeitungsphase stattfinden, wenn man einen Überblick hat und weiß was man vorhat. Im Antrittsvortrag kann man abtasten was

Prof. Snelting von dem Thema hält und wo man Schwerpunkte setzen oder erweitern sollte.

A.3. Abgabe

	Dauer	Umfang
Bachelor	4 Monate	30+ Seiten
Master	6 Monate	50+ Seiten
Studienarbeit	3 Monate	30+ Seiten
Diplom	6 Monate	50+ Seiten

Man kann eine "4.0 Bescheinigung"bekommen, bspw. für die Masteranmeldungen.

Abzugeben sind jeweils 4 gedruckte Examplare der Arbeit, das Dokument als pdf Datei und entstandener Code und andere Artefakte. Außerdem könnten spätere Studenten dankbar sein für TEX-Sourcen.

Zum Drucken empfehlen wir Katz Copy[1] am Kronenplatz, weil wir in Sachen Qualität dort die besten Erfahrungen gemacht haben. Bitte keine Spiralbindung, da sich das schlecht Stapeln lässt. Farbdruck ist nicht verpflichtend, solange in Schwarzweiß noch alle Grafiken lesbar sind.

A.4. Abschlussvortrag

Die Abschlusspräsentation dauert für Bachelorarbeiten 15 Minuten zuzüglich mind. 10 Minuten für Fragen. Bei Masterarbeiten sind 20–25 Minuten für den Vortrag vorgesehen.

Der Vortrag soll innerhalb von vier Wochen nach Abgabe erfolgen, entsprechend Prüfungsordnung. Die Arbeit muss mindestens einen Tag vor dem Abschlussvortrag abgegeben sein, damit sich Prof. Snelting vorbereiten kann.

Am besten direkt im Anschluß den Vortrag ausarbeiten und ein oder zwei Wochen nach Abgabe halten. Der Präsentationstermin muss ein bis zwei Monate im Voraus geplant werden, denn Prof. Snelting hat üblicherweise einen vollen Terminkalender.

[1] `http://www.katz-copy.com/`

A.5. Gutachten

Der Prüfer erstellt ein Gutachten zur Arbeit. Um das Gutachten einzusehen muss ein Antrag beim Prüfungsamt gestellt werden. Der Betreuer bzw. Prüfer darf das Gutachten nur mit genehmigtem Gutachten zeigen. Mündliche Auskunft zur Note ist allerdings möglich.

A.6. Bewertung

- Diplom- und Masterarbeiten *müssen* eine wissenschaftliche Komponente enthalten. Bachelorarbeit *sollten*, aber zum Bestehen ist es nicht notwendig. Wissenschaftlich ist was über reine Implementierungs- bzw. Softwareentwicklungsaufgaben hinausgeht. Üblicherweise findet man theoretische Betrachtungen zu Korrektheit und Effizienz. Willkürliche Daumenregel: Ohne Formel, keine Wissenschaft.

- Diplom- und Masterarbeiten benötigen eigentlich immer Wissen aus dem Diplom- bzw. Masterstudium. Falls das Wissen aus Vordiplom bzw. Bachelor ausreicht, sollte man nochmal darüber nachdenken.

- Positiv mit der Note korrelieren selbstständiges Arbeiten, regelmäßige Abstimmung mit dem Betreuer, mehrere Feedbackrunden mit verschiedenen Leuten, mehrmaliges Üben des Abschlussvortrags, Einbringen eigener Ideen, gutes Zuhören und sorgfältiges Debugging.

- Negativ mit der Note korrelieren wochenlanges Pausieren, Ignorieren von Feedback, Deadlines überziehen und Arbeiten im stillen Kämmerchen.

Disclaimer: Nein, es gibt keinen konkreten Notenschlüssel. Die obigen Punkte sind nur grobe Richtlinien und für niemanden in irgendeiner Weise bindend.

A.7. LaTeX Features

A.7.1. Schriftformatierungen

	serif	sans-serif	fixed-width
normal	Medium **Bold**	Medium **Bold**	`Medium Bold`
italic	*Medium **Bold***	*Medium **Bold***	*`Medium Bold`*
slanted	*Medium Bold*	*Medium **Bold***	*`Medium Bold`*
small-capital	Medium **Bold**	Medium **Bold**	`Medium Bold`

Math fonts: $absXYZ$, absXYZ, $\mathbf{absXYZ}$, absXYZ, absXYZ, $\mathtt{absXYZ}$, and $\mathcal{XYZ}$.

A.7.2. Rand und Platz

Viele Benutzer von LaTeX wollen Ränder und Seitengröße anpassen. Dazu empfehlen wir erstmal die KOMA Script Dokumentation (`koma-script.pdf`) zu lesen, insbesondere Kapitel 2.2. Bevor man mit `\enlargethispage` oder ähnlichen Tricks anfängt, sollte man `\typearea` anpassen.

Falls die Arbeit auf Englisch verfasst wird, sollte man wissen, dass Absätze im Englischen üblicherweise anders formatiert werden. Im Deutschen macht man eine Leerzeile zwischen Absätzen. Im Englischen wird stattdessen die erste Zeile eines Absatzes eingerückt.