Karlsruhe Institute of Technology

Institute for Program Structures
and Data Organization (IPD)

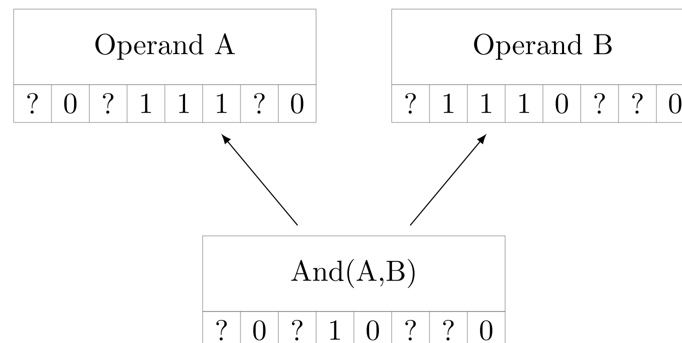
Lehrstuhl Prof. Dr.-Ing. Snelting

Deriving Restrictions on Value Types

Research project

Jonas Fietz

at the faculty of Computer Science



Reviewer: Prof. Dr.-Ing. Gregor Snelting
Advisor: Dipl.-Inform. Sebastian Buchwald

Time: 1st February 2010 – 29th July 2010

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Outline . . . 1

2 Fundamentals **3**
- 2.1 Static Single Assignment Form . . . 3
- 2.2 Firm . . . 3
- 2.3 Mathematical Definitions and Theorems . . . 5
  - 2.3.1 Partially Ordered Set . . . 5
  - 2.3.2 Complete Lattice . . . 6
  - 2.3.3 Ascending Chain Condition . . . 6
  - 2.3.4 Function Properties . . . 6
  - 2.3.5 Fixpoint Theorem on Complete Partial Orders . . . 6
- 2.4 Data Flow Analysis . . . 6
- 2.5 Value Range Propagation . . . 7

3 Design **9**
- 3.1 Walking the Graph . . . 10
- 3.2 Creating the Information . . . 10
  - 3.2.1 Const . . . 10
  - 3.2.2 And . . . 11
  - 3.2.3 Shl . . . 11
  - 3.2.4 Shr . . . 11
  - 3.2.5 Rotl . . . 11
  - 3.2.6 Add & Sub . . . 11
  - 3.2.7 Conv . . . 12
  - 3.2.8 Eor . . . 12
  - 3.2.9 Not . . . 13
  - 3.2.10 Id . . . 13
  - 3.2.11 Phi . . . 13
- 3.3 Complete Partial Order . . . 13
- 3.4 Merging With Existing Information . . . 14
- 3.5 Examples of Optimizations . . . 15
  - 3.5.1 Recognizing Constants . . . 15
  - 3.5.2 Simplify Add If Bits Set Are Disjoint . . . 15
  - 3.5.3 And . . . 15
  - 3.5.4 Optimizing Jump Tables . . . 16

4 Implementation **17**
- 4.1 VRP Struct . . . 17
- 4.2 Derivation of Information . . . 17

5 Evaluation 19

6 Conclusion 21
6.1 Summary 21
6.2 Future Work 21

Bibliography 23

1. Introduction

Compilers today are computer programs used to translate between human understable source languages to computer languages. Usually, they translate some kind of programming language into a target language, typically some kind of assembly or machine code. This enables the development of languages which are easier to grasp for humans by using higher order abstractions.

But often, these kinds of higher order abstractions come with greater target code complexity. To reduce this complexity, compilers typically try to apply heuristics to optimize the code being generated.

Optimized code has multiple advantages, depending on the criteria used for optimization. Typically, it uses less computing power, memory, and runs faster, therefore enabling more efficient use of the existing hardware.

Compiler structure is usually divided in several phases: Analysis, intermediate code generation, optimization and synthesis.

During analysis, the compiler reads the source program and converts it into an intermediate representation. Following this, some generic optimizations are applied to this internal representation. Using this internal representation has the advantage of being target agnostic. The last phase then translates the intermediate representation into code for the target machine.

1.1 Problem Statement

The goal of this project is the implementation of value range propagation in libFIRM, a compiler developed at the Karlsruhe Institute of Technology. Value range propagation (VRP) is usually implemented to derive just information about possible value ranges for each variable in a program. This shall be extended to also include information about bits set or not set for each of the values.

For the optimization to work properly, it needs to terminate, which we will show.

Also, some example optimizations shall be implemented to test the structure used and provide examples.

1.2 Outline

In chapter 2, we introduce the necessary vocabulary to understand this paper. This can be skipped safely by those familiar with Firm, data flow analysis and value range propagation.

In chapter 3, we explain how our implementation of value range propagation has been designed. We demonstrate how we walk the graph and how the information is created for each of the node types. After that, we prove that the described algorithm terminates. To

achieve this, the VRP-information has to be a partially ordered set(section 3.3), and our merging (and updating) function has to be monotone on this set (section 3.4). The chapter is closed with an overlook over a few of the implemented optimizations using VRP.

In chapter 4, we describe the structures created in libFIRM to save the information. Also, we show how the values are initialized and how the information can be accessed using accessor functions created for this purpose.

In chapter 5, we try to evaluate possible performance improvements and see how often our optimizations are actually used.

In chapter 6, we quickly rehash what has been achieved and how one may go about further extending this work and using the newly created infrastructure and information.

2. Fundamentals

This chapter will be used to introduce the terminology needed to explain and show the process of value range propagation as it was implemented in this thesis. We will begin by defining the structures we use in libFIRM, namely the static single assignment form and its derivative Firm.

Then, we will continue to define some mathematical terms, which we will use to proof termination of our algorithm, and explain its workings. This is followed by some remarks about data flow analysis, to later put the process of value range propagation in a context to other analysis tools and optimizations.

2.1 Static Single Assignment Form

Definition: A program is in SSA form, iff every variable is assigned exactly once in the program. In this context, variable means alias-free, local variable.

The static single assignment form is a representation of intermediate languages for compilers. It has been widely accepted and is implemented in most common compilers. SSA's primary characteristic is that each variable is only assigned a value once. For each assignment, a new name is generated for each value. Because of this property, use-def chains are explicit, using SSA simplifies and improves results of various compiler optimizations, due to simpler value attributes.

Obviously, in the case of two parallel control-flow paths, in both of which a variable x is reassigned or defined, one needs a way to deal with the merge point. For this situation, ϕ-nodes were introduced. Assuming that the names assigned to the two instances of x were x_1, x_2, SSA combines them with the ϕ-node, $x_3 = \phi(x_1, x_2)$. The value x_3 represents depends on the control flow path taken.

2.2 Firm

Firm is an intermediate representation (IR) used in the compiler library libFIRM. It is based on the general static single assignment (SSA) form (For further information on libFIRM and Firm, see [Lin02] and [TLB99]). This has the advantage that the definition and uses are directly coupled, which in turn simplifies the results of many compiler optimizations.

When a variable x is assigned a new value, a new version is created for each assignment $(x_1, x_2, \dots)$ and each of its users is adjusted accordingly. In some cases, the value of a variable might depend on the control flow, but each variable can have only one assignment. For this, the mechanism of *Phi*-functions was introduced. So if a variable x_3 might be either x_1 or x_2, then $x_3 = phi(x_1, x_2)$ with a control flow dependent function Φ.

Firm itself is a low level SSA-form. Its graph-based nodes closely resemble the target architecture. Therefore, the IR does not contain a representation of objects or local variables. Also, value numbering can be directly included in the representation. Firm extends the traditional SSA idea by not only modeling the data flow edges, but also the control flow and memory dependencies to include memory accesses in the representation.

Firm operates with 17 different modes for values. These are shown in Table 2.1. They are grouped into several generic modes; these are shown in Table 2.2 (These tables were taken from [TLB99, p.7/8]).

BB:	Blocks	X:	control flow
F:	Floats	D:	Doubles
E:	Exceptions	B_s:	Byte signed
B_u:	Byte unsigned	H_s:	Half word signed
H_u:	Half word unsigned	I_s:	Integer signed
I_u:	Integer unsigned	L_s:	Long signed
L_u:	Long unsigned	C:	Character
P:	Pointer	b:	Boolean
M:	Memory		

Table 2.1: Firm modes

int	$\in Int$
$intb$	$\in Int \cup \{b\}$
int_u	$\in Int \ \wedge \ int_u \ is \ unsigned.$
$float$	$\in Float$
num, num_i	$\in Int \cup Float$
$numP, numP_i$	$\in Int \cup Float \cup Ref$
$data, data_i$	$\in Int \cup Float \cup Char \cup Ref$
$datab, datab_i$	$\in Int \cup Float \cup Char \cup Ref \cup \{b\}$
$dataM$	$\in Int \cup Float \cup Char \cup Ref \cup \{M\}$
$dataMX$	$\in Int \cup Float \cup Char \cup Ref \cup \{M, X\}$
lh	$\in \{BB, M\}$

Table 2.2: Generic modes of operations in Table 2.3

Possible Firm nodes are displayed in table 2.3.

Operation	: Modes of Operands	→ Modes of Results
Block	: X^n	$\to BB$
Start	: BB	$\to X \times M \times P \times P \times data_1 \times \ldots \times data_n \times P$
End	: $BB \times lh_1 \times \ldots \times lh_n$	$\to$
EndReg	: $BB \times lh_1 \times \ldots \times lh_n$	$\to X_{n+1} \times \ldots \times X_{n+m}$
EndExcept	: BB	$\to X_1 \times \ldots \times X_n$
Jmp	: BB	$\to X$
Break	: BB	$\to X$
Cond	: $BB \times b$	$\to X \times X$
Cond	: $BB \times int$	$\to X^n$
Return	: $BB \times M \times data_1 \times \ldots \times data_n$	$\to X$
CallBegin	: $BB \times P$	$\to X^n$

continued on next page

continued from previous page

Operation	: Modes of Operands	→ Modes of Results
Raise	: $BB \times M \times P$	$\rightarrow X \times M$
Const	: BB	$\rightarrow data$
SymConst	: BB	$\rightarrow int$
SymConst	: BB	$\rightarrow P$
Unknown	: BB	$\rightarrow ANY$
Sel	: $BB \times M \times P \times int^n$	$\rightarrow P$
Call	: $BB \times M \times P \times data_1 \times \ldots \times data_n$	$\rightarrow M \times X \times data_{n+1} \times \ldots \times data_{n+m} \times M \times P$
Add	: $BB \times numP \times numP$	$\rightarrow numP$
Add	: $BB \times P \times int$	$\rightarrow P$
Add	: $BB \times int \times P$	$\rightarrow P$
Sub	: $BB \times numP \times numP$	$\rightarrow numP$
Sub	: $BB \times P \times int$	$\rightarrow P$
Sub	: $BB \times int \times P$	$\rightarrow P$
Sub	: $BB \times P \times P$	$\rightarrow int$
Minus	: $BB \times float$	$\rightarrow float$
Mul	: $BB \times int_1 \times int_1$	$\rightarrow int_2$
Mul	: $BB \times float \times float$	$\rightarrow float$
Quot	: $BB \times M \times float \times float$	$\rightarrow M \times X \times float$
DivMod	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int \times int$
Div	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int$
Mod	: $BB \times M \times int \times int$	$\rightarrow M \times X \times int$
Abs	: $BB \times num$	$\rightarrow num$
And	: $BB \times int \times int$	$\rightarrow int$
Or	: $BB \times int \times int$	$\rightarrow int$
Eor	: $BB \times int \times int$	$\rightarrow int$
Not	: $BB \times int$	$\rightarrow int$
Shl	: $BB \times int \times int_u$	$\rightarrow int$
Shr	: $BB \times int \times int_u$	$\rightarrow int$
Shrs	: $BB \times int \times int_u$	$\rightarrow int$
Rot	: $BB \times int_1 \times int_2$	$\rightarrow int_1$
Cmp	: $BB \times datab \times datab$	$\rightarrow b^{16}$
Conv	: $BB \times datab_1$	$\rightarrow datab_2$
Phi	: $BB \times dataM^n$	$\rightarrow dataM$
Filter	: $BB \times dataM^n$	$\rightarrow dataM$
Load	: $BB \times M \times P$	$\rightarrow M \times X \times data$
Store	: $BB \times M \times P \times data$	$\rightarrow M \times X$
Alloc	: $BB \times M \times int_u$	$\rightarrow M \times X \times P$
Free	: $BB \times M \times P$	$\rightarrow M$
Sync	: $BB \times M^n$	$\rightarrow M$

Table 2.3: Syntax of Firm operations. For resolution of generic modes *data* etc. see Table 2.2

2.3 Mathematical Definitions and Theorems

In the following subsections, we define terminology needed for proofs later on.

2.3.1 Partially Ordered Set

Given a set L, a relation $\sqsubseteq \subseteq L \times L$ is called partial order, iff it has the properties

1. Reflexivity: $\forall l : l \sqsubseteq l$

2. Transitivity: $\forall l, m, n : l \sqsubseteq m \wedge m \sqsubseteq n \Rightarrow l \sqsubseteq n$
3. Anti-Symmetry: $\forall l, m : l \sqsubseteq m \wedge m \sqsubseteq l \Rightarrow l = m$

Given a subset $M \subset L$, $l \in L$ is an *upper bound*, if $\forall l' \in M : l' \sqsubseteq l$, and l is a *lower bound*, if $\forall l' \in M : l \sqsubseteq l'$. A least upper bound l is an upper bound of M that also satisfies $l \sqsubseteq l_0$, whenever l_0 is another upper bound of M. The greatest lower bound is defined analogously. Subsets M of a partially ordered set do not need to have least upper or greatest lower bounds, but if they exists, they are unique and are denoted $\sqcup M$ and $\sqcap M$ respectively.

A complete partial order is a partially ordered set M, which has a smallest element $\bot$ and in which every ascending chain $C \subseteq M$ has an upper bound $sup(C)$. (Definition from [NNH99])

2.3.2 Complete Lattice

A complete lattice is a partially ordered set L, such that all subsets have least upper and greatest lower bounds, and that $\bot = \sqcup L$ is the least element and $\top = \sqcap L$ is the greatest element. (Definition from [NNH99].)

2.3.3 Ascending Chain Condition

An ascending chain is a sequence of elements $a_1, a_2, a_3, \dots$ of a partially ordered set M, so that $a_1 \leq a_2 \leq a_3 \dots$. Therefore every two elements in C are comparable. Therefore, if the chain istaken as a set, it isa completely ordered set.

A partially ordered set satisfies the ascending chain condition (ACC), iff every ascending chain becomes stationary, i.e. $\exists n \in \mathbb{N} : a_n = a_{n+1} = a_{n+2} = \dots$.

2.3.4 Function Properties

A function $f : O_1 \rightarrow O_2$ between two complete partial orders is called monotone, if $x \leq y \Rightarrow f(x) \leq f(y)$.

A function f between two complete partial orders O_1, O_2, $f : O_1 \rightarrow O_2$ is called continuous if it maps a chain $C_1 \subseteq O_1$ to a chain $C_2 \subseteq O_2$, while preserving their suprema.

$$f(sup(C)) = sup(f(C)), with f(C) = f([x_1, x_2, \dots]) = [f(x_1), f(x_2), \dots]$$

A continuous function is always monotone. (Definition from [DP02])

2.3.5 Fixpoint Theorem on Complete Partial Orders

For a complete partial order $(O, \leq)$ and a monotone function $f : O \rightarrow O$, there exists a least, unique fixpoint X with f(X) = X.

For a complete partial order $(O, \leq)$, the smallest element $\bot$, and a continuous function $g : O \rightarrow O$, there exists a least, unique fixpoint $X = sup\{f^n(\bot)\}$, which can be calculated iteratively.

For the latter there also is the dual theorem: For a complete partial order $(O, \geq)$, largest element $\top$, and a contiuous and falling function $f : O \rightarrow O$, there exists a largest, unique fixpoint $X = inf\{f^n(\top)\}$, and it can be calculated iteratively. (See [DP02] for details)

2.4 Data Flow Analysis

Data-flow analysis (DFA) is a type of program analysis. It is a static analysis at compile time, which attempts to gather information about the visibility and availability of data at each point in a computer program and the dependencies resulting thereof. These analyses include for example reaching definition analysis, available expressions analysis or the detection of dead code. Data flow analyses are usually divided in two categories: forward analyses, using the control flow graph, and backward analysis, using the reverse control flow graph.

Commonly, DFA operates on the control flow graph (CFG), as it tries to obtain information for each point in a program. The information to be derived is usually defined in

an abstract way on a complete lattice. DFA also usually operates on basic blocks in the CFG. For these, we view the entry and exit states, defined as

$$exit_b = trans_b(entry_b)$$

$$entry_b = join_{p \in pred_b}(exit_b)$$

for a block b. Here, $join$ is the join-Operation in the complete lattice, therefore creating the entry state for the next block. The more important function is the transfer function $trans$, which describes the effect of block b on the entry state. This way, we get a set of solvable equations. The resulting values for entry and exit may then be used to get the information desired. Backward analysis is defined analogously.

There are two common solutions for the equation set. The first and simpler one is an iterative algorithm which initializes all entry values with known good values and then iteratively applies the transfer function for each block and then the join operation to get new entry values. (It has to be said that in our case, we do not need the mentioned join operation due to the structure of Firm.) This process continues until there are no more changes in these values. This algorithm terminates as long as the combination of transfer and join function is monotonic with respect to this complete lattice. It can even be shown that this also applies to complete partial orders, which we will use for the value range propagation.

The second algorithm for solving the equations is the worklist algorithm. This will also be the one we use for VRP. The work-list algorithm does not indiscriminately iterate over all the blocks and uses a worklist instead. The worklist is initialized through one iteration over the whole graph as in the first algorithm. If the exit state of a block changes, all successors which have already been visited are added to the worklist, if not already in there. After the initial iteration is over, we process each block from the worklist. In case its exit state changes, all successors are added back to the worklist. For performance reasons, the successors are usually only added if they are not yet in the list.

For further information in this matter, please refer to [NNH99].

2.5 Value Range Propagation

Value Range Propagation (VRP) is, traditionally, a technique to determine bounds on the ranges of values assumed by variables at specific points in programs. This technique can be used to eliminate unneccessary tests, verify correct operation, choose more appropriate data representation, to name just a few examples. It was first described by W.H. Harrison in [Har77].

The version described by Harrison includes two parts for the analysis, range propagation and range analysis. Range propagation is used to derive and propagate refinements in the accuracy of range information due to the structure and data in the program. Range analysis on the other hand tries to track changes to variables in loops, thereby deriving stepping information. The latter has the downside of requiring symbolic representation and calculation, which in most cases will be adequate for proofs, but not for a generic compiler library. [Pat95] used very simple representations only able to represent basic arithmetics with one unknown.

Therefore, in contrast to the original paper, we chose to simplify the process by removing the range analysis part, thereby also avoiding the need for symbolic representation of value equations. But we extended the original design with two different ideas; for one, we are not only saving range information, but we also derive anti-ranges, which represent ranges which can not be assigned to a variable. Anti-ranges are also implemented in GCC and LLVM, two of the more popular open compilers. The idea for this originally came from [Pat95].

As a second extension, we also decided to include information about bit patterns for each value. Analogously to the way ranges are propagated, we also propagate information

about the state of each bit for a value, meaning if it is known to be true, false or if it is unknown.

3. Design

There are several ways to go about designing value range propagation, but all are based on the same principles. Most of these principles also apply to constant propagation, which is a very well-understood principle in the field of compiler optimizations. It is based on the knowledge of certain values at compile time and therefore, some expressions can be evaluated at compile time rather than as runtime through propagation the values through the graph. This may save a lot of processing later during program runs. Constant propagation itself is a simple forward data flow problem, and the principles behind value range propagation are, as mentioned, fairly similar.

What VRP tries to achieve is that instead of propagating exact values, we try to propagate, as the name implies, ranges for the variables. We extend this idea by trying to derive information about the bits of each value. So for each variable, we try to find lower and upper limits for its range as well as the information which bits are definitely set or not set.

An extension to this idea would be not deriving only value ranges, but all possible values. If one were to extend the data structures used so that all possible subsets of V were representable, one could operate on a complete lattice. This approach gives the additional value that one would find the minimal fixpoint possible. But as this would require a data structure able to save every combination of values, it would possibly have to use huge amounts of memory and is therefore not feasible in a real-world compiler.

Mathematically, we try to derive some knowledge about the variable as a point in a partially ordered set $= (V; \subseteq)$, where the $\emptyset$ is defined as non-derivable. To achieve this goal, we save three pieces of information for each node.

- **Value Range** $R(U, L, T)$: The range itself is defined as an upper (U) and a lower (L) bound. There are two possible types (T) the range may have, referred to as *range* and *anti-range*. Range has the expected meaning; anti-range means that the value can not be within the range defined by the boundaries. The range type can also be *undefined*, implying that no information is yet known, or *varying*, indicating no meaningful information can be derived, or the information is not representable with our limited data types.

- **Bits Set (B)**: These are the bits that are definitely set, e.g. through an or-operation. A 1 refers to a bit that is set, a 0 might be set.

- **Bits Not Set ($\overline{B}$)**: These are the bits that are definitely not set, analogous to "Bits Set", such that 1 refers to a bit that might be set, and 0 refers to a bit that is definitely not set.

If we thing about the information saved in the bit vectors for each variable, we actually have ternary values: bit is certainly set, bit might be set, bit is certainly not set. To represent this, we used two bit vectors in our implementation. Two bit vectors bring some redundancy with them, but we chose this path because of the ease of implementation and later usage.

The semantics of the bit vectors were defined this way, because writing several optimizations and derivations for information hinted in the direction that this definition would result in easier expressions. One easy example is trying to test for a constant value: If the bit sets are equal, we know the exact value. But this decision is based on only a small subset of possible expressions, so other definitions might be better.

To improve the readability, we will assume that the operations *and* ($\wedge$) and *or* ($\vee$) on bit vectors are all bitwise, having their semantics from boolean logic. So with $B_i(x)$ referring to the i-th bit in the bit vector $B(x)$, we define $B(a) = B(b) \wedge B(c)$ as $\forall i : B_i(a) = B_i(b) \wedge B_i(c)$.

In the following subsections, we will detail the specific steps needed to walk the graph, determine the value ranges through the operands of the node, merge new values with ones already existing and prove the termination of the algorithm.

3.1 Walking the Graph

Value range propagation is a typical forward data-flow problem. There are lots of ways to walk a graph, but for this kind of problem, one uses a reverse postorder iteration on the graph. It walks the nodes in a topological order, and is the most efficient order, as each node is visited before its successors, the only exception being those having backedges, which is exactly the direction of the data-flow.

Due to the existence of circles, caused by loops, jumps, or other control flow operations, not all information can be derived in a single walk, no matter which order. So in the case that a successor node has already been visited during the initial graph walk (this is only the case, if there is some kind of circle in the graph), then we add it to a work list for later reconsideration. This is necessary, as one of its predecessors has more accurate information now, so we might be able to derive more accurate information for this node as well.

Afterwards, we just continue using the typical work list algorithm for data-flow problems from section 2.4. For every node in the work list, we calculate the results from its predecessors. In the case that its VRP information changes, we add all its successors back to the work list. Because we know that the VRP problem has a fixpoint, the algorithm will terminate (See section 3.3).

3.2 Creating the Information

The creation of new range information can be understood on a fairly intuitional base. To guarantee correctness, all nodes are initialized so that they allow all possible values. This means, that the range-type is initialized to *UNDEFINED*. B is set to all zeroes, $\overline{B}$ is set to all ones.

The correctness of the implemented types is now shown exemplary for a few of these types. Also, we include a few sample graphs to explain the process. For all examples, we assume that no prior VRP information existed, and that the data type has 16 bits. When talking about the information derived, any information not mentioned is left at the default values.

3.2.1 Const

We begin with the simplest case: The *Const*-node. It should be immediately obvious how to set all the values:

With V being the value of the const, the range $R = (V, V, \text{RANGE})$, the bit vectors are $B = \overline{B} = V$.

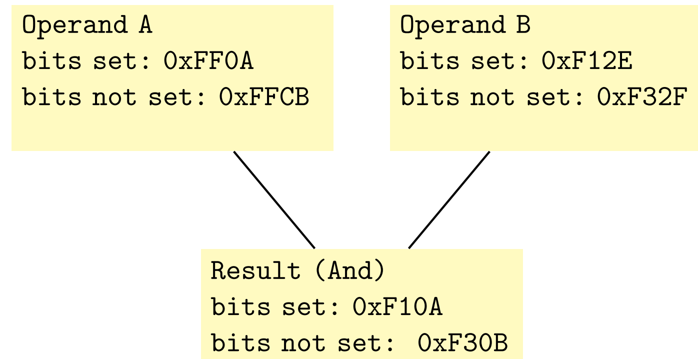


Figure 3.1: Example: Creating the information for an *And*-node.

3.2.2 And

Deducing range information for *And*-nodes is only possible in a few, very specific cases. Therefore, we did not consider these. But of course, as with all the bit operations, it is fairly easy to derive the bit vectors.

Suppose we have the two operands a and b of the node n, with $B(x)$, $\overline{B}(x)$ as defined in chapter 3. From this follows that $B(n) = B(a) \wedge B(b)$ and $\overline{B}(n) = \overline{B}(a) \wedge \overline{B}(b)$ (Also see example Figure 3.1).

Or-nodes can be derived accordingly.

3.2.3 Shl

For *Shl*-nodes and all the other shift operations, we only derive information for constant shift factors. These should be the most common ones, and come with little implementation cost. Additionally, we only need to look at correctness for a shift factor of one, as all other factors can be determined inductively.

To deduce the information, take the bit vectors of the left operand and shift it according to the constant value of the right operand. This operation shifts in zeros from the right. This is correct, as with any value being shifted, it neccessarily has zeroes shifted in as well. Therefore our bit vectors stay correct.

3.2.4 Shr

For all shifting operations, the “old” bits are obviously still true, if they are shifted by the same amount. Therefore we have to consider the newly introduced bits. For the arithmetic right shift, the sign-bit is shifted in from the right. In the case that the value for this node has a one, the corresponding bit in the bits set vector can be either 1 or 0, implying we already know that there is a one or that there may be a one. If we shift right by one now, the two left-most bits are either both 1, which is correct, or both 0, implying we do not know if the bits at the two left-most positions are 1 or 0. This is correct.

For the bits not set vector we only have to view the case where the left-most bit is 1, as 0 would implicate that the left-most bit of the value is 0. If it is 1, and we do a right-shift, both left-most bits are 1, implying that both left-most bits of the value can be one. This is correct.

The case for the value being non-negative can be handled analogously.

3.2.5 Rotl

Rotl is a circular shift. As the bits only change their position, applying the same operation to both bit vectors is correct.

3.2.6 Add & Sub

For *Add*-nodes, we only handle the obvious cases. E.g. overflows are discarded, giving us no information. Some implementations, such as GCC’s, also calculate values for overflows, but this is due to the C-specification defining the results as unknown in several cases, so

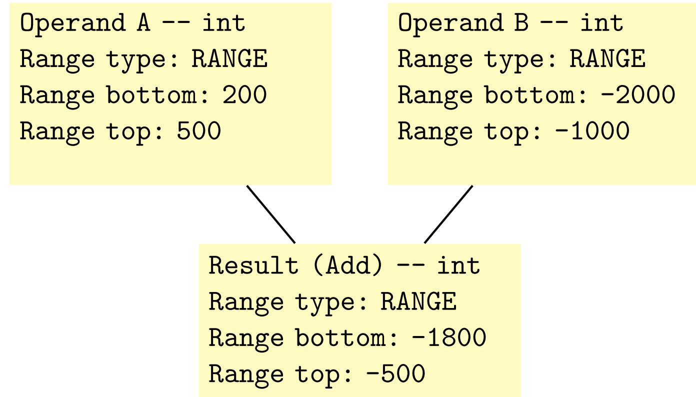


Figure 3.2: Example: Creating the information for an *Add*-node, no overflow

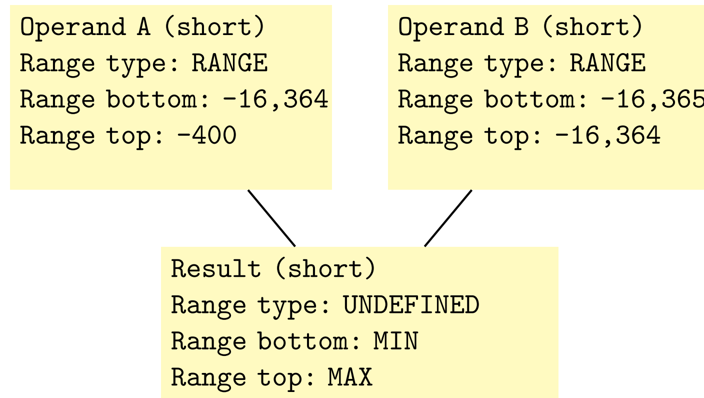


Figure 3.3: Example: Creating the information for an *Add*-node. Negative overflow both for top and bottom (short int typically has a size of two bytes, thus a range from -16,384 to 16,383). In this example, one can easily see that even when overflows were defined, deriving information for all cases would get quite complex.

that optimization is feasible. As LibFIRM has multiple independent front-ends, such a behavior is neither possible nor desired.

In the case that both operands have a defined VRP range, we just add both top values, and both bottom values, to get the new values respectively, but only use these values if no overflow occurred (Also see examples Figure 3.2, Figure 3.2.6). *Sub*-nodes are handled accordingly.

3.2.7 Conv

Conv-nodes convert between different value types, i.e. between different possible sizes. So there are two cases: smaller type to bigger type or bigger type to smaller type. In the first case, we just embed the bit vectors in new ones of the target size. All newly added bits are set to their unknown states. As the old ranges still hold true, we just change their type and keep the information from the operand. In the second case, we cut all bits from the bit vectors, which are not needed anymore. For the ranges, we check if they are within the prior borders. If this is the case, we keep them, otherwise they are set to their respective type limits.

3.2.8 Eor

For *Eor*-nodes, it follows:

$$B(n) = (B(a) \land \neg\overline{B}(b)) \lor (B(b) \land \neg\overline{B}(a))$$
$$\overline{B}(n) = \neg[(B(a) \land B(b)) \lor (\neg\overline{B}(a) \land \neg\overline{B}(b))]$$

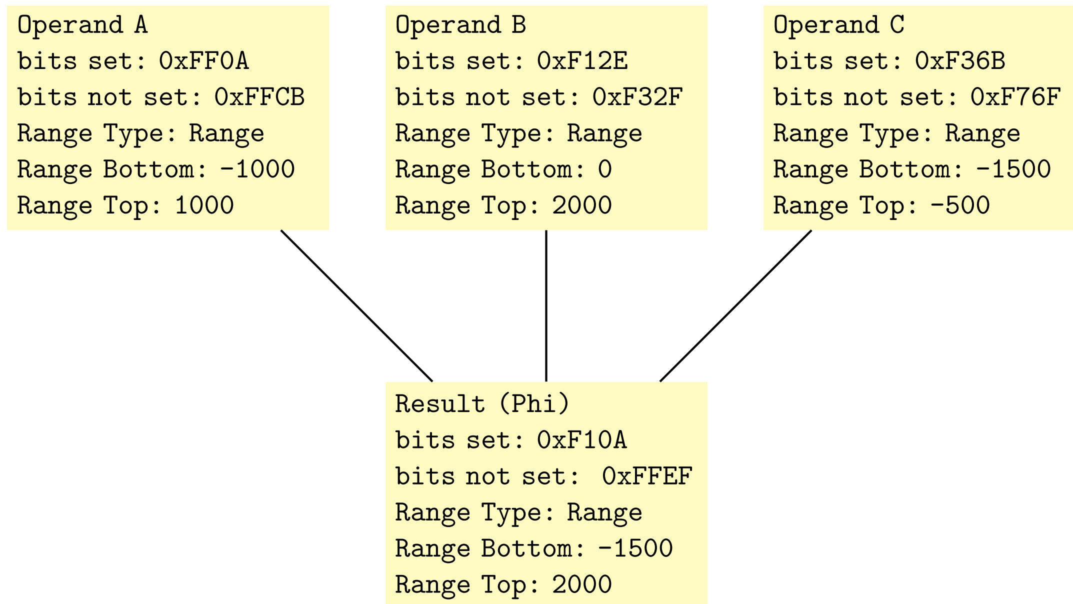


Figure 3.4: Example: Creating the information for a *Phi*-node.

3.2.9 Not

For *Not*-nodes, we invert the bit-vectors of the operand.

3.2.10 Id

For *Id*-nodes, we use the information from the operand.

3.2.11 Phi

At *Phi*-nodes, information is combined, so any of the operands information might hold true. So we have to do the opposite of merging and we can only use the information which all operands have. We use *and* for the bits set, *or* for the bits not set, and, assuming that all prior nodes have ranges associated, we use the maximal range suggested by all their limits, because the real value could come from any of the operands.

Examples are shown in Figure 3.4 and Figure 3.5.

3.3 Complete Partial Order

As explained in section 2.4, to show that our algorithm terminates, the information on which we operate has to be a complete partial order. We also explained that the combination of the transfer and the join function normally has to be monotonic to guarantee termination. Our algorithm is somewhat special in that it preserves data over multiple analysis phases, merging old data with new data. This merge function is also called every time data is updated.

Normally, one would show that all our transfer functions from section 3.2 are monotone, but we can skip this step as it suffices to show that the VRP algorithm works on a partially ordered set with a limited height, and that the function merging the information for each node is monotone. We prove the latter in section 3.4.

As our merge function only narrows the ranges, we actually produce a descending chain. To use the dual theorem from subsection 2.3.5, we have to show that our ranges and bitsets are a complete partial order with a largest element $\top$ and that every descending chain has an infimum. Because our merge function is defined piecewise for ranges and bit vectors, it suffices to show this seperately for each.

Our ranges are a subset of $(\mathcal{P}(V), \subseteq)$, which is a partially ordered set. Therefore, they are a partial order O, which can be defined as

$$L(I(V); \subseteq)$$

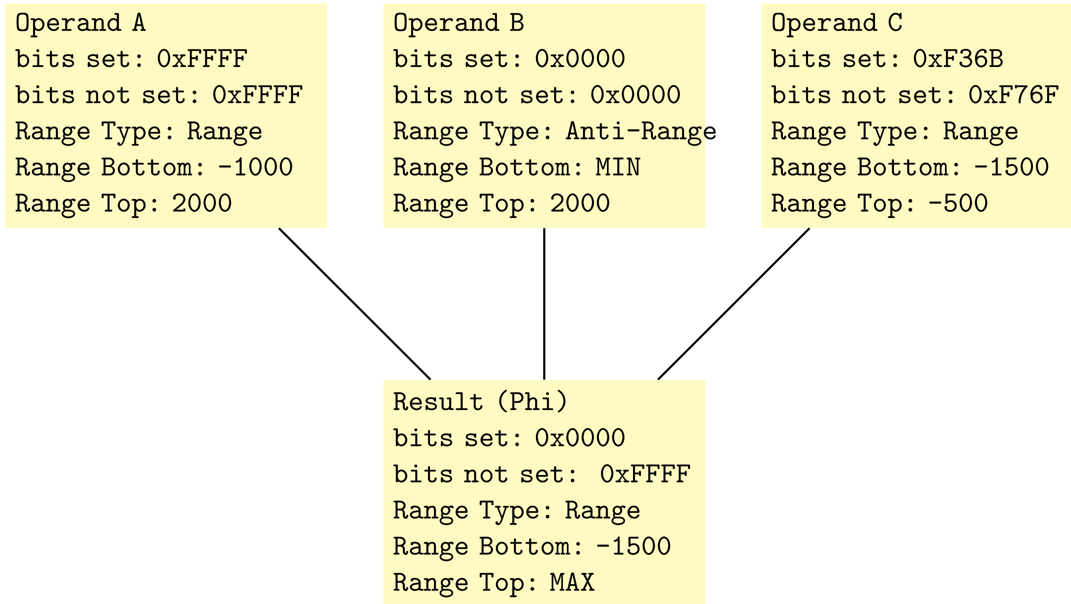


Figure 3.5: Example Two: Creating the information for a *Phi*-node.

with

$$I(V) = \{[a,b] | a,b \in V\} \cup \{M | M = V \setminus [a,b]; a,b \in V\}$$

$\top$ is obviously the complete V. For every descending chain, it has an infimum defined as the intersection, which will be a single value in the best case. One could think that the minimum were the empty set, but this would mean that the variable could not take any value, which is impossible in a valid program graph.

Boolean algebras are always complete lattices. The power set of any set S is also a boolean algebra with the operations union and cut. The bit vectors can be seen as the characteristic functions of the power set, so the bit vectors are also a complete lattice. As every complete lattice is also a complete partial order, we have shown what we needed. The maximum in our case would be B equal to all zeroes, and $\overline{B}$ equal to all ones, meaning we have no information yet.

3.4 Merging With Existing Information

As we said earlier, our implementation of value range propagation is special in the context of libFIRM, because opposed to most analyses it preserves the information over multiple analysis phases. As changes to the graph structure itself can occur due to different phases, multiple runs of the analysis might make sense to derive more information from this new structure. To conserve as much information as possible, the old information has to be combined with the newly derived by a merge function. The merging process is explained in the following paragraphs. The merging function also combines the information during multiple iterations, so as explained in section 3.3, to proof that the algorithm terminates, we need to show that our merge function is monotone and descending. As explained above, we will show this piecewise for both bit vectors and the ranges.

The solution for the bit vectors follows intuitively. Both have to hold true, so we use binery *or* for the information for the bits set, and we use *and* for the bits not set. The reason for two differing operations is immediately obvious if one considers the semantics, where a 1 in B means known information, while a 1 in $\overline{B}$ means unknown information. Beginning with an initialization of B to all zeroes, only setting zeroes to ones, we obviously have a descending and monotone function. The analog is true for $\overline{B}$.

For ranges, we try to use the information which is stricter, i.e. the interval which is smaller. There are three cases, which we will discuss separately:

- For two ranges, we use the higher minimum bound and lower maximum bound. That way, our interval can only shrink.
- For two anti-ranges, we use the lower minimum bound and the higher maximum bound, so our new interval becomes smaller.
- If we have a range and an anti-range, we try to combine these. If the values deemed possible overlap, we create a new range containing exactly this overlap. In the case that the anti-range is contained in the range, we cannot use this information due to the way we represent range information, as this situation would actually imply two possible ranges. Therefore, one has to decide which information to keep. We chose to use the information already present.

As the interval can only get smaller than the prior interval, we have a descending, monotone function. As the merging process as a whole is a descending and monotone function, our algorithm terminates.

The theoretical complexity of our algorithm depends largely on the height h of our partial orders. As shown, each node is visited a maximum of h times. Depending on the view, h can be seen as an, albeit huge, constant. This leads to a theoretical complexity of $O(n)$, where n is the number of nodes in the Firm graph. Our runtime is usually much lower than suggested by h. For DFA, there are usually two ways to go about finding a fixpoint. One would be initializing our ranges to the empty set, and then enumerating every possible value by iterating over the graph until there are no more changes. This would require completely unrolling loops, although one could find ways around it. But these work-arounds imply a loss of precision, exactly what we tried to prevent in the first place. As explained in section 2.5, these are known problems of the range analysis.

The other way is to initialize the ranges to all possible values and removing values which cannot be taken. This has a much better runtimes, and was therefore the way chosen.

3.5 Examples of Optimizations

In this section, we will present some of the optimizations we implemented, using the information derived by VRP, to give the reader an idea of what is possible. As examples, we implemented some optimizations originally developed for constant values which we can now also evaluate if we have enough information.

3.5.1 Recognizing Constants

In some situations, we might be able to derive ranges containing only one value, or we might have $B = \overline{B}$, meaning all bits are known. In this case, we have a constant value and can replace this node by a constant node, enabling further optimizations.

3.5.2 Simplify Add If Bits Set Are Disjoint

If we add two numbers, and both have disjoint bits set (meaning that only one of them has a bit set at a certain position), then we may replace this *And* by an *Or*. This enables us to apply distributivity, opening the chance for further optimizations.

Example: We have two nodes, a and b. Now have have an *Add* with them as operands: $c = Add(a, b)$. If we have $\overline{B_a} = 0xFC08$ and $\overline{B_b} = 0x02FF$, then we know that their bit sets are disjoint, meaning that adding them will never result in a carry-bit. Therefore, *Or* is semantically equivalent to *Add* for these two nodes.

3.5.3 And

Suppose that we have an *And* on two operands a, b, where a is constant. If we know that b is zero everywhere where a is zero or unknown, then we know that $c = And(a, b) = b$, because c can only be 1 where a is one, and is certainly zero where a is zero. As b is zero where a is zero, all other zeroes are solely dependent on b, as a is one at those places anyway. So we can safely replace the *And* by b.

Example: $a =??111?01, b = 000?100?, And(a, b) = 000?100?$.

3.5.4 Optimizing Jump Tables

Especially in automatically generated jump tables, there might be a lot of cases which can never be taken. To enable this optimization, we use the VRP information for the selector of the *Cond*-node. If it has any range information, we test if our projection value falls within this range. The same goes for the bit patterns. In case that the selector has bits set which are not set in our projection value, or vice versa, we can assume that this projection will never occur and can therefore remove it and replace this *Proj*-node with a *Bad*-node.

4. Implementation

This chapter describes the concrete implementation as created for this project and implemented in libFIRM.

4.1 VRP Struct

VRP-information for each node is saved in a struct *vrp_attr*.

```
typedef struct {
  int valid;              /**< This node has valid vrp information*/
  tarval *bits_set;       /**< The bits which by analysis are
                               definitely set.
                               0: may be not set, 1: definitely set*/
  tarval *bits_not_set; /**< The bits which by analysis are
                               definitely not set.
                               1: may be set, 0: definitely not set*/

  /**< The range type represented by range_top, range_bottom */
  enum range_types range_type;

  tarval *range_bottom, *range_top;
} vrp_attr;
```

Our *range_type* is an enum, defined as follows:

```
enum range_types {
        VRP_UNDEFINED, /**< No information could be derived so far */
        VRP_RANGE,      /**< bottom and top form a range,
                             including both values */
        VRP_ANTIRANGE, /**< range from bottom to top can not be
                             assigned, but borders might be */
        VRP_VARYING     /**< information can not be derived */
};
```

4.2 Derivation of Information

To save our information, we use the above struct, which is saved in the *phase* utilities provided by libFIRM. This approach allows us to automatically execute initialization code for each node upon first access to the relevant VRP information.

We initialize the nodes with valid data, B set to all zeroes, $\overline{B}$ set to all ones, and the range type to UNDEFINED. Range top and range bottom are both set to the maximum of their respective type, but as the range type is always checked first, this does not carry

any meaning yet. *vrp->valid* is used as a safety measure against accessing uninitialized memory in the case of undiscovered bugs.

LibFIRM has an abstraction called *tarval*s, which provides an abstraction from the concrete implementation of calculations and sizes of variables for the target architecture. We use the *tarval* functions to get adequately sized variables for each value type, as well as to do calculations on the values.

```
vrp->range_type = VRP_UNDEFINED;
vrp->valid = 1;
vrp->bits_set = get_mode_null(mode);
vrp->bits_not_set = get_mode_all_one(mode);
vrp->range_bottom = get_tarval_top();
vrp->range_top = get_tarval_top();
```

For safety, if a node for which VRP-information cannot be derived is accessed, we set all its values to *get_tarval_bad()*.

We also implemented an accessor function for the VRP information, for e.g. the various optimizations using the information.

```
vrp_attr *vrp_get_info(const ir_node *n);
```

This function returns a pointer to the information for the node *n*, or it returns *NULL* if an error occured and no information exists yet. This way, to access information, we can use this syntax:

```
vrp_attr *vrp = vrp_get_info(n);
if (vrp && INSERT_CONDITIONS_TO_CHECK)
```

5. Evaluation

In this section, we evaluate the performance of the VRP-implementation presented. This work focused more on the implementation of the analysis rather than implementing a lot of optimizations using the VRP information. Therefore, we did not expect great performance improvements in any benchmark.

We did measure our performance comparing it to the performance of libFIRM without the VRP information. As those numbers mostly do not show a measurable difference, we also counted how often our optimizations were applied for each of the programs compiled. To get numerical data on how good our current optimizations perform, we chose to use the test programs from the SPEC CINT2000 collection (see [Sta]), part of CPU2000. Originally, CPU2000 was used for performance measurement for a wide range on hardware. Due to its nature, it can also be used as a compiler test platform, when comparing programs—compiled with different versions—on the same platform. In our case, we tested on a Core2Duo E5300@2.6Ghz and 4GB of RAM, running Debian Linux with Kernel Version 2.6.31-17, with enabled Physical Address Extension.

To achieve meaningful results, the tests were run four times each, once with VRP activated and once without it. The results are shown in Table 5.1.

Test	novrp	vrp	Relative Runtime	# optimizations
164_gzip	103	103	99.4%	14
175_vpr	94	94	99.9%	8
176_gcc	49	49	99.8%	619
181_mcf	98	97	99.3%	0
186_crafty	52	53	100.6%	9
197_parser	124	124	99.8%	8
253_perlbmk	82	83	100.5%	0
254_gap	60	58	97.4%	385
255_vortex	110	110	99.6%	12
256_bzip2	97	97	99.5%	5
300_twolf	116	114	98.8%	10
		Average performance	99.5%	

Table 5.1: Performance measurements comparing libFIRM with and without VRP activated

As one can see, although there are variations in the runtime, a small improvement is measurable. To see whether the improvements were measurement inaccuracies or caused by the optimizations, we also counted the optimiziations using the VRP information.

The high number of applied optimizations for the GCC test case are caused by the huge code base, but it seems like most of that code is not actually used in the test case. The results for GAP show that we had at least some success. GAP is an acronym for groups, algorithms and programming, and is a test case in the SPEC suite that heavily depends on integer calculations for combinatorics, so it is exactly the kind of program for which we would expect performance increases.

6. Conclusion

6.1 Summary

With this thesis, we created the basic architecture and structures neccessary in libFIRM to support value range propagation. Based on this, we implemented derivations to get information from the existing SSA graphs and nodes. We extended on the idea of value range propagation as it is usually implemented by extending the normal range information for each value to also include bit patterns representing further known information. At last, we implemented a few basic optimizations on this, showing how to use this newly created information.

6.2 Future Work

Based on this work, there are lots of features and improvements which can be made. The gained VRP information can be used in many other optimization and analysis parts of the compiler. Some examples, such as loop optimization, come to mind. Using this example, one can show some of those possibilities: Within loops, one may know due to VRP that a loop may only be executed once, or maybe that the iterator is always even, that the iterator does not exceed a certain number x, or many other things.

Another great field of optimizations certainly lies in the optimizations of pointers. Especially for languages with automated array bounds checking, this could bring huge performance improvements as suggested by [BvEG04].

We implemented a few basic optimizations already, as shown in section 3.5. As one can see, a lot of these optimizations are fairly trivial, but their number will probably greatly improve their success.

Also, the analysis part could be further improved through more complex derivations, or more fine tuned analysis. Loosing some of the genericity, one could for example implement an adder for two's complement. Although libFIRM supports target architectures using different binary representations, this would bring additional benefit on this prevalent architecture. Additionally, one could find ways to use more of the node types as we currently do.

Finally, as suggested in section 3.4, we currently do not save multiple derived ranges, so it might make sense to extend our structures in this regard (as done in [Pat95]), although other major implementations, such as GCC or LLVM, also do not do this.

Bibliography

[BvEG04] J. Birch, R. van Engelen, and K. Gallivan, “Value Range Analysis of Conditionally Updated Variables and Pointers,” in *Proceedings of Compilers for Parallel Computing*, 2004, pp. 265–276. [Online]. Available: http://www.cs.fsu.edu/~engelen/cpcpaper.pdf

[DP02] B. A. Davey and H. A. Priestley, *Introduction to lattices and order / B.A. Davey, H.A. Priestley.* Cambridge University Press, Cambridge [England] ; New York :, 2002.

[Har77] W. Harrison, “Compiler analysis of the value ranges for variables,” *IEEE Transactions on Software Engineering*, vol. 3, pp. 243–250, 1977.

[Lin02] G. Lindenmaier, “libFIRM – A Library for Compiler Optimization Research Implementing FIRM,” Tech. Rep. 2002-5, September 2002. [Online]. Available: http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps

[NNH99] F. Nielson, H. R. Nielson, and C. L. Hankin, *Principles of Program Analysis*. Springer, 1999, second printing, 2005.

[Pat95] J. R. C. Patterson, “Accurate static branch prediction by value range propagation,” 1995.

[Sta] Standard Performance Evaluation Corporation, “CINT2000, part of CPU2000 test by SPEC,” http://www.spec.org/cpu2000/CINT2000/.

[TLB99] M. Trapp, G. Lindenmaier, and B. Boesler, “Documentation of the intermediate representation firm,” Universität Karlsruhe, Fakultät für Informatik, Tech. Rep. 1999-14, Dec 1999. [Online]. Available: http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz