

Binding Time Analysis for Bits

CS-263 Course Project

Eylon Caspi

May 25, 1998

Abstract

Using a high level language (HLL) to specify a computation for synthesis in ASIC or FPGA hardware requires aggressive compiler analysis to capture bit-level program characteristics that cannot normally be expressed in a HLL but which are important to specializing and optimizing logic for size and speed. It is useful, for instance, to identify unchanging bits of a variable and subsequently remove using partial evaluation any logic which depends on them. We describe a binding-time analysis for bits which can identify static and dynamic bits of computed expressions. The analysis proves to be expensive, possibly taking logarithmically as many steps as the actual binary computation.

1 Introduction

In the quest for ever-faster computation, custom computing machines can offer performance superior to traditional microprocessors. ASIC and FPGA implementations of custom logic continue to become more attractive with improvements in semiconductor manufacturing and reconfigurable architectures. The computational power of custom logic derives primarily from the ability to specialize a computation down to the bit-level, using application-specific, minimum-bit-width data paths and such optimizations as logic-level constant-propagation to trim away unneeded logic. Specialized logic can thus typically be made smaller and/or faster than general purpose alternatives such as a fixed-width ALU in a microprocessor.

Designing custom logic, however, is far from easy. Currently-available technologies such as hardware description languages (HDL) and schematic capture require a designer to specify a computation in complete bit-level detail. This requires significant expertise and effort for logic design and debugging, and is highly prone to design error.

A more attractive programming solution would be to use a familiar high-level programming language (HLL). The structured syntax and word-based computational model of HLLs conveniently abstract away the details of a logic implementation, allowing a programmer to focus instead on function, as well as providing semantic safeguards such as type-checking. A word-model of computation is certainly appropriate for programming a microprocessor, where a pre-fabricated, fixed-width ALU can perform operations of various bit-widths at fixed cost. The purpose of custom logic, however, is to make smaller and faster implementations which avoid the compromises of such a multi-purpose data path.

Unfortunately, the word-model of HLLs usually makes it difficult or impossible to express the bit-level characteristics that a custom-logic implementation could exploit and optimize. In C, for instance, quantized word-widths (char, int, long) and the lack of a boolean variable type are obvious culprits. It is impossible to express in C, for example, that an integer variable is range-limited and needs only 10 significant bits, or that it is quantized to multiples of 16 and never changes its 4 least-significant bits, or that squaring it can hence be accomplished using a 6x6-bit multiplier.

Recent empirical findings in U.C.Berkeley's BRASS¹ group suggest that such undiscovered bit-level characteristics are frequent and lead to significant wasted computation. In a variety of C programs from such sources as SPEC '95 and the UCLA MediaBench suite [4], program profiling has found that typically 50% of bit-level read-accesses to program variables retrieve bits that never change during the course of a program run. Furthermore, typically over 90% of these bit-reads are to unchanging high- or low-order bit-ranges of variables. Since every read-access represents an input to a computation, these unchanging bits of storage represent significant wasted computation, due primarily to excessively-wide word operations. While some of the waste is due to dynamic program behavior and cannot be foreseen at compile-time, those bit-level characteristics which can be known at compile-time are still seldom expressible by a programmer in a HLL such as C.

We would like to develop compiler analysis techniques to discover certain bit-level characteristics which are not expressible in C and which lead to wasted computation. Discovering unused high-order bits of integral quantities can be accomplished by known techniques of value range analysis [2] [1], which computes via data-flow analysis a range $[l, u]$ of possible values for every numerical expression in a program. The purpose of such an analysis in microprocessor-oriented compiler technology is primarily to predict the outcome of conditional statements involving numerical comparisons. Value range analysis cannot capture unused low-order bits or constant bits embedded in the middle of variables. Also, because it focuses on arithmetic operations, value range analysis does not discover constant bits in bit-parallel logic operations (operators $\&$, $|$, \sim in C).

¹Berkeley Reconfigurable Architectures, Systems, and Software

The typical compiler hammer for exploiting unchanging quantities is binding-time analysis (BTA) followed by partial evaluation. The goal of BTA is to label every expression and statement in a program as either static (known at compile-time) or dynamic. This is typically done using data-flow analysis with the simple lattice domain: $unknown \sqsubset static \sqsubset dynamic$. A partial evaluator can then remove any computation known to have a static result, computing and directly substituting that result into program statements that use it.

In this paper, we explore the extension of binding-time analysis to the bit domain. In section 2 we discuss an abstract interpretation suitable for propagating bit staticness information in a data-flow analysis. In section 3 we discuss a framework for bit-level compiler analysis based on the word-level analysis of Hornof and Noyé [3]. We do not discuss the details of bit-level partial-evaluation, since it involves significant computer arithmetic and, in practice, would likely be tied to a logic synthesis engine.

2 Bit Binding-Time Propagation

2.1 Development of an Abstract Interpretation

Binding-time analysis for bits seeks to determine, for each bit of every computed expression, whether that bit’s value can or cannot be known at compile-time. In binding-time analysis for words, it is easy to propagate a 3-valued judgment $unknown \sqsubset static \sqsubset dynamic$ through every HLL program statement. It is not possible to use this lattice directly on bits because it is not trivial or even decidable for some word operations how the staticness of an arbitrary input bit affects the staticness of an arbitrary output bit. We do know how the binary value of an input bit affects the value of an output bit – this is binary arithmetic and logic. It is plausible, then, to extend the 3-valued lattice by splitting the *static* category into separate 0 and 1 cases in order to be able propagate a bit-level lattice value through any word-level operation. The resulting 4-valued lattice \mathcal{B}_4 is shown in figure 1. The computation of bit binding-times can then be done using abstract interpretation on a word-level domain \mathcal{D}_4 which is a direct product of bit-level lattices \mathcal{B}_4 with pointwise least-upper-bound (ex. $\mathcal{D}_4 = \mathcal{B}_4^{32}$ for 32-bit quantities).

Abstract interpretation is a framework for estimating a computation by performing an analogous computation in a simpler, abstract domain. Estimating integer arithmetic may involve an original domain which is the integral range of computable expressions (ex. $2^{-31} \dots (2^{31} - 1)$ for 32-bit signed arithmetic) and an abstract domain of subsets thereof such as positive/negative, even/odd, etc. An abstract interpretation would involve propagating an abstract estimate through every operation from the original program. If the abstract domain is a lattice — a complete partial order with least-upper-bound (LUB, \sqcup) and greatest-lower-bound (GLB, \sqcap) defined for every pair — then the abstract com-

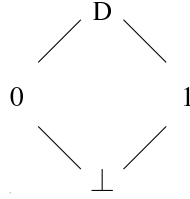


Figure 1: Simple domain for bit-level binding-time analysis. D represents a dynamic bit; 0 and 1 represent static bits; \perp represents an unknown bit binding-time.

putation can be performed iteratively using a data-flow framework. Such an implementation repeatedly visits the computation steps in some order, retaining for each desired solution the LUB with all its previous approximations, until a fixed point is reached.

One unconventional feature of our proposed abstract interpretation is that the abstract domain $\mathcal{D}_4 = \mathcal{B}_4^N$ (for N -bit-wide computation) is in fact larger, not smaller, than the original domain $\{0, 1\}^N$. This seems to imply that the abstract computation is harder than the original one. We will see, however, that the nature of the lattice allows iterated abstract computations to converge faster than the original program executes operations.

2.2 Example

Consider the following C code fragment:

```
int i=0;
while (i<64)
    i=i+8;
```

Let $\alpha(i)$ denote the abstract value of i , representing the set of values which i may take during program execution. Immediately after initialization, $\alpha(i) = (00000000)_b$ (shown here as if i had only 8 bits). Abstract interpretation will iterate through the loop, each time computing a new abstract value for i by adding $\alpha(8) = (00001000)_b$ to the previous $\alpha(i)$, and retaining the LUB of the two as the new $\alpha(i)$. The first addition changes one bit of $\alpha(i)$, updating it to be:

$$\begin{aligned}
 \alpha(i) &\leftarrow \alpha(i) \sqcup (\alpha(i) \bar{+} \alpha(8)) \\
 &= (00000000)_b \sqcup ((00000000)_b \bar{+} (00001000)_b) \\
 &= (00000000)_b \sqcup (00001000)_b \\
 &= (0000D000)_b
 \end{aligned}$$

where $\bar{+}$ is the abstract addition operator, and D is the dynamic abstract bit, whose concrete counterpart may be 0 or 1. A subsequent iteration updates $\alpha(i)$

to be:

$$\begin{aligned}
\alpha(i) &\leftarrow \alpha(i) \sqcup (\alpha(i) \bar{\sqcup} \alpha(i)) \\
&= (0000D000)_b \sqcup ((0000D000)_b \bar{\sqcup} (00001000)_b) \\
&= (0000D000)_b \sqcup (000DD000)_b \\
&= (000DD000)_b
\end{aligned}$$

A third iteration updates $\alpha(i)$ to $(00DDD000)_b$. Assuming the `while` condition can be handled intelligently, iteration converges there, concluding that i has exactly 3 dynamic bits. In this example, the abstract computation requires logarithmically as many steps as the actual computation, since rather than simply counting, each iteration propagates an abstract D to the next significant bit.

2.3 A Better Abstract Domain

While the abstract domain D_4 is sufficient for unsigned arithmetic, it is too weak for signed arithmetic. Consider a signed variable x which takes value ± 1 in the following code fragment:

```

int x;
if (cond)
    x=1;
else
    x=-1;

```

After evaluating the `if` statement, abstract interpretation finds:

$$\begin{aligned}
\alpha(x) &\leftarrow \alpha(1) \sqcup \alpha(-1) \\
&= (00000001)_b \sqcup (11111111)_b \\
&= (DDDDDD1)
\end{aligned}$$

Thus the analysis determines that x has 7 dynamic bits. These 7 high-order bits are in fact sign-extension bits and all change together. There is no way, however, to represent such information in the purely bit-parallel domain D_4 . Hence the analysis cannot see that x nominally uses only 1 dynamic bit, namely a sign bit.

To handle signed arithmetic, we expand the abstract domain to explicitly represent sign-extension bits. The new 7-valued lattice \mathcal{B}_7 for bitwise staticness is shown in figure 2. Note the added representations S for a dynamic sign bit and S_0, S_1 for static sign bits. The new word domain \mathcal{D}_7 is a direct product \mathcal{B}_7^N with pointwise LUB and the additional constraint that abstract sign bits appear only in a single, maximal run of identical abstract bits in the most-significant position (ex., $S_0S_0S_00$ or $S_0S_0S_10$ are not allowed). It may be necessary to use

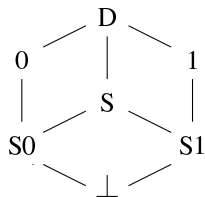


Figure 2: Signed domain for bit-level binding-time analysis. D =dynamic bit, S =dynamic sign-extension bit, $0, 1$ =static bits, S_0, S_1 =static sign-extension bits, \perp =unknown bit binding-time.

both signed and unsigned abstract interpretation together, converting between the two when a C type-cast or type-promotion calls for it. Conversions between abstract domains are discussed below.

Doing abstract computations in the signed domain is more difficult than in the unsigned domain, in part because the lattice is larger, and in part because the consistency of sign-extension bits must be maintained. Consider, for instance, a naive computation of abstract bit-parallel AND (denoted $\bar{\wedge}$) using parallel abstract bitwise ANDs (denoted $\bar{\wedge}_b$). If we take the abstract AND of a static bit $(0,1)$ with a static sign-extension bit (S_0, S_1) to be a regular static bit (which is higher in the \mathcal{B}_7 lattice than a static sign-extension bit), the following may happen:

$$(S_0 S_0 S_0 S_0)_b \wedge_b (S_1 S_1 S_1 0)_b = (S_0 S_0 S_0 0)_b \quad (\text{bad!})$$

It is easy enough, nevertheless, to perform abstract computations in the unsigned domain \mathcal{D}_4 , then convert into the signed domain \mathcal{D}_7 by setting sign-extension bits appropriately. Indeed, in the unsigned domain, the naive computation of abstract bit-parallel AND using parallel abstract bitwise ANDs is correct. Conversion into the signed domain involves scanning an abstract value from MSB towards LSB, converting the first contiguous run of 0 or 1 into a run of S_0 or S_1 , respectively. Hence we can implement a signed abstract computation as follows:

- convert each argument into the unsigned word domain \mathcal{D}_4 , mapping $S_0 \rightarrow 0$, $S_1 \rightarrow 1$, $S \rightarrow \{0, 1\}$, creating a set of 2 unsigned abstract word values;
- for each combination of unsigned arguments (up to 2 for unary operators, 4 for binary operators), compute in the unsigned domain using the bitwise operations of \mathcal{B}_4 ;
- convert each result into the signed domain \mathcal{D}_7 and keep their LUB.

2.4 Abstract Operations

We now describe in detail the computations in the unsigned abstract domain \mathcal{D}_4 . The computations are implemented using parallel or cascaded abstract bitwise

operations from \mathcal{B}_4 in a manner which mimics binary arithmetic and boolean logic circuits. Simple bit-parallel logic operations such as AND can be performed directly using parallel bitwise operations. Computations which require a carry-chain or reduction tree in traditional boolean logic implementations, for instance addition or comparison, can be implemented using abstract carry chains.

Note that in operations in \mathcal{B}_4 , an unknown binding-time \perp acts much the same as a dynamic binding-time D . It does observe near-strictness as a bottom element, nevertheless, in that a computation with \perp in any input which produces a non-static output (not 0,1) should be taken as outputting \perp rather than D . For the sake of brevity, we omit computations with input \perp in some of the discussion below. Also, for visual clarity in the tables, we write \top in place of D .

Shifts Abstract shifts are a straightforward extension of boolean logic shifts. Arithmetic shifts duplicate the MSB, while logical shifts push in a static $0 \in \mathcal{B}_4$

Bit-parallel logic The abstract computation of bit-parallel AND, OR, XOR, and NOT is easily performed by parallel application of the following bit operators:

$\bar{\wedge}_b$	\perp	0	1	\top
\perp	\perp	0	\perp	\perp
0	0	0	0	0
1	\perp	0	1	\top
\top	\perp	0	\top	\top

$\bar{\vee}_b$	\perp	0	1	\top
\perp	\perp	\perp	1	\perp
0	\perp	0	1	\top
1	1	1	1	1
\top	\perp	\top	1	\top

$\bar{\oplus}_b$	\perp	0	1	\top
\perp	\perp	\perp	\perp	\perp
0	\perp	0	1	\top
1	\perp	1	0	\top
\top	\perp	\top	\top	\top

$\bar{\neg}_b$	
\perp	\perp
0	1
1	0
\top	\top

Boolean logic The abstract manipulation of boolean values with C operators $\&\&$ and $\|\|$ can be done using the bit logic operators from \mathcal{B}_4 shown above. Note that words may be reduced to boolean values only through comparison operators, discussed below.

Equality/inequality comparisons Abstract numerical comparisons can be implemented by a carry chain which propagates the abstract value of the comparison result from LSB towards MSB. A bit-slice on the carry chain would have the result tables shown below for abstract operators $\bar{<}$, $\bar{\leq}$, $\bar{>}$, $\bar{\geq}$, $\bar{\neq}$, $\bar{=}$. Note that the first 4 inequality operators below are correct only for unsigned comparisons. In each table, the LSB column lists output for the LSB slice; the next 3 columns list output for propagating slices.

		$a <_b b$						$a \leq_b b$			
a	b	LSB	$\leq_{b,h}^{in} = 0$	$\leq_{b,h}^{in} = 1$	$\leq_{b,h}^{in} = \top$	a	b	LSB	$\leq_{b,h}^{in} = 0$	$\leq_{b,h}^{in} = 1$	$\leq_{b,h}^{in} = \top$
0	0	0	0	1	\top	0	0	1	0	1	\top
0	1	1	1	1	1	0	1	1	1	1	1
0	\top	\top	\top	1	\top	0	\top	1	\top	1	\top
1	0	0	0	0	0	1	0	0	0	0	0
1	1	0	0	1	\top	1	1	1	0	1	\top
1	\top	0	0	\top	\top	1	\top	\top	0	\top	\top
\top	0	0	0	\top	\top	\top	0	0	\top	\top	\top
\top	1	\top	\top	1	\top	\top	1	1	\top	1	\top
\top	\top	\top	\top	\top	\top	\top	\top	\top	\top	\top	\top

		$a >_b b$						$a \geq_b b$			
a	b	LSB	$\geq_{b,h}^{in} = 0$	$\geq_{b,h}^{in} = 1$	$\geq_{b,h}^{in} = \top$	a	b	LSB	$\geq_{b,h}^{in} = 0$	$\geq_{b,h}^{in} = 1$	$\geq_{b,h}^{in} = \top$
0	0	0	0	1	\top	0	0	1	0	1	\top
0	1	0	0	0	0	0	1	0	0	0	0
0	\top	0	0	\top	\top	0	\top	\top	0	\top	\top
1	0	1	1	1	1	1	0	1	1	1	1
1	1	0	0	1	\top	1	1	1	0	1	\top
1	\top	\top	\top	1	\top	1	\top	1	\top	1	\top
\top	0	\top	\top	1	\top	\top	0	1	\top	\top	\top
\top	1	0	0	\top	\top	\top	1	0	\top	\top	\top
\top	\top	\top	\top	\top	\top	\top	\top	\top	\top	\top	\top

		$a \neq_b b$						$a \equiv_b b$			
a	b	LSB	$\neq_{b,h}^{in} = 0$	$\neq_{b,h}^{in} = 1$	$\neq_{b,h}^{in} = \top$	a	b	LSB	$\equiv_{b,h}^{in} = 0$	$\equiv_{b,h}^{in} = 1$	$\equiv_{b,h}^{in} = \top$
0	0	0	0	1	\top	0	0	1	0	1	\top
0	1	1	1	1	1	0	1	0	0	0	0
0	\top	\top	\top	1	\top	0	\top	\top	0	\top	\top
1	0	1	1	1	1	1	0	0	0	0	0
1	1	0	0	1	\top	1	1	1	0	1	\top
1	\top	\top	\top	1	\top	1	\top	0	\top	\top	\top
\top	0	\top	\top	1	\top	\top	0	\top	0	\top	\top
\top	1	\top	\top	1	\top	\top	1	\top	0	\top	\top
\top	\top	\top	\top	1	\top	\top	\top	0	\top	\top	\top

Addition Abstract Two's complement addition can be performed by cascaded abstract full-adders with a least-significant carry-in of 0:

$\bar{\top}b$		$c_{in} = 0$		$c_{in} = 1$		$c_{in} = \top$	
a	b	s	c_0	s	c_0	s	c_0
0	0	0	0	1	0	\top	0
0	1	1	0	0	1	\top	\top
0	\top	\top	0	\top	\top	\top	\top
1	0	1	0	0	1	\top	\top
1	1	0	1	1	1	\top	1
1	\top	\top	\top	\top	1	\top	\top
\top	0	\top	0	\top	\top	\top	\top
\top	1	\top	\top	\top	1	\top	\top
\top	\top	\top	\top	\top	\top	\top	\top

Multiplication Abstract multiplication can be computed as a sum of partial products. Using the notation $a\#n$ to mean a sign-extended to n bits (still in the unsigned domain \mathcal{B}_4):

$$a\bar{*}b = \sum_{i=0}^{N-1} (a_i\#2N)\bar{\wedge}((b\#2N)\ll i)$$

This computation is sufficient for multiplying 2 unsigned numbers or one signed times one unsigned but requires some modification to multiply two signed numbers. A variety of techniques for binary number multiplication,

for instance Booth recoding, can be extended to the abstract domain in order to perform this multiplication.

Division Unfortunately, there are no reasonably easy integer division algorithms which may be extended to work in the abstract domain. Quotient magnitude can, however, be estimated using dividend and divisor magnitudes, thus capturing the staticness of the quotient’s high-order bits. Specifically, if dividend a has its most-significant non-sign bit (MSNSB) in position α , and divisor b has its MSNSB in position β , then quotient a/b has its MSNSB in position $\gamma \leq (\alpha - \beta)$. The quotient’s sign-extension bits can be taken as $sign_a \oplus sign_b$, while all lower-order bits can be conservatively approximated to be dynamic. If enough static bits of a and b are known, a factorization algorithm might try to recover the staticness of the quotient’s low order bits, should a and b have common factors.

Modulo Like division, modulo remaindering lacks any easy algorithm to extend to the abstract domain but is amenable to an upper-bound approximation. Specifically, we know that $(a \bmod b) < b$, so the remainder can be taken to have as many sign-extension bits as b , with all lower-order bits being dynamic.

2.5 Abstract Inverse Operations

A context-sensitive binding-time analysis is one which allows variables to have different binding-times at different program points. For instance, a variable x may be known to have all static bits after initialization by a constant, then have dynamic bits inside a loop, and again have all static bits at loop exit for a loop such as `while (x<10)`. A context-sensitive analysis can be made more precise if it can extract from a conditional expression a separate estimate of variable binding-times for each conditioned branch. For instance, it is useful to know that after exiting a loop `while (x & 0xF0)`, variable x obeys:

$$(S \dots S0000 \perp \perp \perp \perp)_b \sqsubseteq x \sqsubseteq (S \dots S0000 \top \top \top \top)_b.$$

Complementary or more specific information may already be available about x , in which case it could be LUB-ed with information gathered from the conditional expression.

One way to extract binding-times from conditionals is with inverse abstract operators. For a boolean equation in \mathcal{B}_4 , $(e_1 \text{ op } e_2) = e_3$, the inverse operator op^{-1} seeks to estimate the binding-time of e_1 from those of e_2 and e_3 . The inverse for boolean logic operators and bit-parallel logic operators is fairly easy to handle using abstract-bit operations. We take the inverse operator’s estimate for e_1 as the GLB of all e_1 values for which the equation $(e_1 \text{ op } e_2) = e_3$ holds. Such a least estimate can be safely LUB-ed with other information about e_1 during binding-time propagation. In the tables below, top-row labels represent

the expression output e_3 , side-row labels represent input e_2 , and table contents represent the inverse operator's estimate for input e_1 .

$\overline{\wedge}_b^{-1}$	\perp	0	1	\top
\perp	\perp	0	\perp	\perp
0	\perp	\perp	\perp	\perp
1	\perp	0	1	\top
\top	\perp	0	\perp	1

$\overline{\vee}_b^{-1}$	\perp	0	1	\top
\perp	\perp	\perp	1	\perp
0	\perp	0	1	\top
1	\perp	\perp	\perp	\perp
\top	\perp	\perp	1	0

$\overline{\oplus}_b^{-1}$	\perp	0	1	\top
\perp	\perp	\perp	\perp	\perp
0	\perp	0	1	\top
1	\perp	1	0	\top
\top	\perp	\perp	\perp	\perp

$\overline{-}_b^{-1}$	\perp	0	1	\top
	\perp	1	0	\top
	\perp	1	0	\top

Abstract numerical-comparison operators (ex. $\overline{<}$, $\overline{=}$) are defined using carry chains and can be inverted using reverse carry chains which propagate from MSB towards LSB. Consider the inverse operator $\overline{<}^{-1}$ which, given $(A\overline{<}B) = e$, estimates $A \in \mathcal{D}_4$ from $B \in \mathcal{D}_4$ and the boolean outcome $e \in \mathcal{B}_4$. A slice of the inverse carry chain must map abstract bits $(b, \overline{<}_{out})$ to an estimate of the original slice's inputs, $(a, \overline{<}_{in})$. The result table for such a slice is as follows, each input estimate being the GLB of all corresponding inputs which generate the desired comparison outcome in the original slice:

b	$\overline{<}_{out}$	a	$\overline{<}_{in}$
\perp	\perp	\perp	\perp
\perp	0	1	0
\perp	1	0	1
\perp	\top	\perp	\perp
0	\perp	\perp	\perp
0	0	\perp	\perp
0	1	0	1
0	\top	\perp	1
1	\perp	\perp	\perp
1	0	1	0
1	1	\perp	\perp
1	\top	1	0
\top	\perp	\perp	\perp
\top	0	1	0
\top	1	0	1
\top	\top	\perp	\perp

The inverse abstract operation $\overline{<}^{-1}$ could be used, for instance, to compute that inside the loop `while (x<10)`, the abstract value of an unsigned variable x is lower-bounded by $(0\dots 0\perp\perp\perp\perp)_b \in \mathcal{D}_4$, or equivalently by $(S_0\dots S_0\perp\perp\perp\perp)_b \in \mathcal{D}_7$. To see this, let $B = \alpha(10) = (0\dots 01000)_b \in \mathcal{D}_4$ and apply the carry-chain operator $\overline{<}_b^{-1} : (b, \overline{<}_{in}) \rightarrow (a, \overline{<}_{out})$ on successive abstract bits of B , from MSB towards LSB. The inside of the loop represents the case when $(x < 10)$ is true,

so the computation begins with $\bar{z}_{out} = 1$ in the most-significant place. Proceedings towards the LSB, \bar{z}_b^{-1} produces $(a, \bar{z}_{in}) = (0, 1)$ until the 1-bit of B is encountered, at which point $(b, \bar{z}_{out}) = (1, 1)$ produces $(a, \bar{z}_{in}) = (\perp, \perp)$, and all subsequent carry-chain applications again yield $(a, \bar{z}_{in}) = (\perp, \perp)$.

Inverse abstract operators can be developed in a similar fashion for all abstract operators, including carry-chain based integer arithmetic. The abstract-value estimate of such inverse operators is not always effective, however. For instance, the \bar{z}^{-1} operator presented above computes that, upon exiting the loop `while (x<10)`, the abstract value of x is lower-bounded by the uselessly-weak quantity $(\perp \dots \perp \perp \perp \perp)_b$. All the numerical comparison operators are similarly weak in this analysis framework, yielding the bottom abstract value in one of their two branches. As such, the high cost and weak yield of inverse abstract operators may make their use impractical. To better handle numerical inequality comparisons, we might combine bit binding-time analysis with value-range analysis [2] [1] whose representation of contiguous value ranges is better suited for such comparisons.

3 Compiler Analysis Framework

The abstract interpretation described in the previous section could be used in a number of compiler analysis framework. In this section we describe how it could be used in a particular context-sensitive data-flow analysis framework. This framework is based on the word-level binding-time analysis engine of Hornof and Noyé [3], which assigns to each expression and statement of a C program a binding-time in the lattice: *unknown* \sqsubset *static* \sqsubset *dynamic*. We retain most of that engine’s high-level structure but alter certain low-level data-flow formulas to compute expression binding-times with bit granularity (in \mathcal{D}_7) rather than word granularity.

The binding-time analysis is based on a notion of program state which represents the binding-times for program variables. The data-flow analysis computes binding-times for expressions and statements which depend directly on these program variables, iteratively updating the program state. The analysis framework we describe is context-sensitive, meaning that it allows for variables to have a different binding-time at different points in a program. This requires maintaining a separate state for each program point. We define a “program point” to be associated with a statement in a restricted subset of C where a statement is the smallest syntactic entity which can side-effect program variables (by assignment). The C subset we consider is shown in figure 3. It involves some restrictions to control-flow and assignment forms which simplify the analysis. Note that *uop* and *bop* respectively represent unary and binary arithmetic/logical operators. The unary address-of and dereferencing operators are handled separately.

```

exp      ::=  const | id | &lexp | *exp
          |   lexp bop exp | uop exp
lexp     ::=  id | *exp
stmt     ::=  lexp=exp
          |   if (exp) stmt else stmt
          |   do stmt while (exp)
          |   {stmt*}
          |   id(exp*)
          |   id=id(exp*)
          |   return exp
          |   return
type-spec ::= int | char | *type-spec
decl     ::=  type-spec id
func-def ::=  type-spec id(decl*) stmt
program  ::=  decl* func-def*

```

Figure 3: The subset of C considered by our analysis, mostly from [3].

Specifically, we define for each program statement s the pre-state $\llbracket s \rrbracket_{\text{in}}$ and post-state $\llbracket s \rrbracket_{\text{out}}$. Formally, a state $\llbracket s \rrbracket : \mathcal{L} \rightarrow \mathcal{D}_7$ is a function from program locations \mathcal{L} (i.e. variables) to corresponding binding-times \mathcal{D}_7 , or equivalently a set of ordered pairs in $\mathcal{L} \times \mathcal{D}_7$. Data-flow equations describe the relation between each statement’s pre-state and post-state based on the operations of that statement. The equations can be solved iteratively to find all program states, simultaneously computing expression and statement binding-times consistent with those states.

The data-flow equations relating program states, as given by [3], are shown in figure 4. The state transfer functions $t_s()$ and $t_{s,e}()$ account for state updates in assignment statements and control-flow statements, respectively. Their formal definitions are given in figure 5. Note the use of functions $\text{defs}(s)$ and $\text{unambiguous-defs}(s)$ which compute the set of locations possibly-defined and unambiguously-defined by statement s , respectively. These functions derive from prior liveness analysis and pointer alias analysis. In the language of [3], a location is unambiguously-defined by an assignment if the location is not aliased, i.e. if no more than one location can be affected by the assignment.

The statement/expression binding-time propagation functions used by the state transfer functions $t_s()$, $t_{s,e}()$ are shown in figure 6. Several departures are taken from the analysis of [3]. Statement binding-times remain in the 3-valued domain $\text{unknown} \sqsubset \text{static} \sqsubset \text{dynamic}$, while expression binding-times are in the new domain \mathcal{D}_7 . The binding-time propagation functions for expressions consisting of unary or binary arithmetic/logic operations now use the abstract computations described in section 2.4, denoted by $\text{uop-bt}(e, \Sigma)$ and $\text{bop-bt}(e_1, e_2, \Sigma)$.

This analysis, as it stands, does not incorporate binding-time estimation us-

$$\begin{aligned}
& \text{lexp}^{e_1 =^s \text{exp}^{e_2}} : \llbracket s \rrbracket_{out} = t_s(\llbracket s \rrbracket_{in}) \\
\text{if}^s (\text{exp}^e) \text{stmt}_1^{s_1} \text{ else } \text{stmt}_2^{s_2} : & \llbracket s_1 \rrbracket_{in} = \llbracket s \rrbracket_{in} \\
& \llbracket s_2 \rrbracket_{in} = \llbracket s \rrbracket_{in} \\
& \llbracket s \rrbracket_{out} = t_{e,s_1}(\llbracket s_1 \rrbracket_{out}) \sqcup t_{e,s_2}(\llbracket s_2 \rrbracket_{out}) \\
\text{do}^s \text{stmt}^{s_0} \text{ while } (\text{exp}^e) : & \llbracket s_0 \rrbracket_{in} = \llbracket s \rrbracket_{in} \sqcup t_{e,s_0}(\llbracket s_0 \rrbracket_{out}) \\
& \llbracket s \rrbracket_{out} = \llbracket s_0 \rrbracket_{out} \\
\{\text{stmt}_1^{s_1} \dots \text{stmt}_n^{s_n}\} : & \llbracket s_1 \rrbracket_{in} = \llbracket s \rrbracket_{in} \\
& \llbracket s_{i+1} \rrbracket_{in} = \llbracket s_i \rrbracket_{out}, 1 \leq i < n \\
& \llbracket s \rrbracket_{out} = \llbracket s_n \rrbracket_{out} \\
\text{return}^s \text{exp}^e : & \llbracket s \rrbracket_{out} = \llbracket s \rrbracket_{in}
\end{aligned}$$

Figure 4: Data-flow equations for program states, from [3].

$$\begin{aligned}
t_s(\Sigma) &= \{(loc, \text{stmt-bt}(s, \Sigma)) \mid loc \in \text{defs}(s)\} \sqcup (\Sigma \setminus \text{unambiguous-defs}(s)) \\
t_{e,s}(\Sigma) &= \{(loc, \text{exp-bt}(e, \Sigma)) \mid loc \in \text{defs}(s)\} \sqcup \Sigma
\end{aligned}$$

Figure 5: State transfer functions from [3], denoting an update of state Σ in statement s via (i) assignment, (ii) control-flow conditioned on expression e . $\Sigma \setminus S$ denotes the resetting to \perp of binding-times in state Σ for those variables in set S .

ing inverse abstract operators. Adding this analysis would involve modifying the data-flow equations for control-flow statements so that the pre-state of each conditioned block becomes a LUB of the statement pre-state and an estimate derived from the conditional expression. Specifically, the equations for **if-else** and **do-while** statements would be changed as in figure 7. The new state transfer function $\tau_e(\Sigma)$ computes a state estimate from Σ using inverse operators, assuming expression e were true, as described in section 2.5.

4 Conclusion

We have proposed a binding-time analysis for the bits of a computation described in a HLL, and discussed a context-sensitive data-flow framework for performing the analysis on a subset of the C programming language. The analysis is not cheap, as it involves abstract interpretation in a domain larger than the computed expressions themselves. We have seen that a data-flow computation in this domain may require logarithmically as many steps as the actual binary computation. Furthermore, the analysis requires initial passes for liveness analysis and pointer alias analysis. While such an analysis may be justified in some circumstances for enabling the use of a HLL for synthesizing hardware,

- statements:

$$\begin{aligned}
& \text{lexp}^{e_1} =^s \text{exp}^{e_2} : \text{stmt-bt}(s) = \text{lexp-bt}(e_1, \llbracket s \rrbracket_{in}) \sqcup \text{exp-bt}(e_2, \llbracket s \rrbracket_{in}) \\
& \text{if}^s (\text{exp}^e) \text{stmt}_1^{s_1} \text{ else } \text{stmt}_2^{s_2} : \text{stmt-bt}(s) = \text{exp-bt}(e, \llbracket s \rrbracket_{in}) \sqcup \text{stmt-bt}(s_1) \sqcup \text{stmt-bt}(s_2) \\
& \text{do}^s \text{stmt}^{s_0} \text{ while } (\text{exp}^e) : \text{stmt-bt}(s) = \text{exp-bt}(e, \llbracket s_0 \rrbracket_{out}) \sqcup \text{stmt-bt}(s_0) \\
& \{ \text{stmt}_1^{s_1} \dots \text{stmt}_n^{s_n} \} : \text{stmt-bt}(s) = \sqcup_{1 \leq i < n} \text{stmt-bt}(s_i) \\
& \text{return}^s \text{exp}^e : \text{stmt-bt}(s) = \text{exp-bt}(e, \llbracket s \rrbracket_{in})
\end{aligned}$$

- expressions:

- right-hand expressions:

$$\begin{aligned}
& \text{const}^e : \text{exp-bt}(e, _) = \text{const} \\
& \text{id}^e : \text{exp-bt}(e, \Sigma) = \text{lookup}(\Sigma, \text{id}) \\
& \&^e \text{lexp}^{e_0} : \text{exp-bt}(e, \Sigma) = \text{lexp-bt}(e_0, \Sigma) \\
& *^e \text{exp}^{e_0} : \text{exp-bt}(e, \Sigma) = \text{exp-bt}(e_0, \Sigma) \sqcup (\sqcup_{loc \in \text{aliases}(e)} \text{lookup}(\Sigma, loc)) \\
& \text{exp}_1^{e_1} \text{ bop}^e \text{exp}_2^{e_2} : \text{exp-bt}(e, \Sigma) = \text{bop-bt}(e_1, e_2, \Sigma) \\
& \text{uop}^e \text{exp}^{e_1} : \text{exp-bt}(e, \Sigma) = \text{uop-bt}(e, \Sigma)
\end{aligned}$$

- left-hand expressions:

$$\begin{aligned}
& \text{id}^e : \text{lexp-bt}(e, \Sigma) = \perp \\
& *^e \text{exp}^{e_0} : \text{lexp-bt}(e, \Sigma) = \text{exp-bt}(e_0, \Sigma)
\end{aligned}$$

Figure 6: Binding-time propagation functions for statements and expressions, mostly from [3].

$$\begin{aligned}
& \text{if}^s (\text{exp}^e) \text{stmt}_1^{s_1} \text{ else } \text{stmt}_2^{s_2} : \llbracket s_1 \rrbracket_{in} = \llbracket s \rrbracket_{in} \sqcup \tau_e(\llbracket s \rrbracket_{in}) \\
& \llbracket s_2 \rrbracket_{in} = \llbracket s \rrbracket_{in} \sqcup \tau_{\neg e}(\llbracket s \rrbracket_{in}) \\
& \llbracket s \rrbracket_{out} = t_{e, s_1}(\llbracket s_1 \rrbracket_{out}) \sqcup t_{e, s_2}(\llbracket s_2 \rrbracket_{out}) \\
& \text{do}^s \text{stmt}^{s_0} \text{ while } (\text{exp}^e) : \llbracket s_0 \rrbracket_{in} = \llbracket s \rrbracket_{in} \sqcup t_{e, s_0}(\llbracket s_0 \rrbracket_{out}) \sqcup \tau_e(\llbracket s_0 \rrbracket_{out}) \\
& \llbracket s \rrbracket_{out} = \llbracket s_0 \rrbracket_{out} \sqcup \tau_{\neg e}(\llbracket s_0 \rrbracket_{out})
\end{aligned}$$

Figure 7: Data flow equations for program states, modified to include binding-time estimates using abstract inverse operators.

it may be too slow for programming rapid-prototyping environments such as FPGAs.

The empirical finding that most of a C program's reads of unchanging bits come from the upper or lower bit-ranges of variables rather than from the middle suggests that we should seek a cheaper analysis which concentrates solely on the binding-time of these extreme bit-ranges. The relative speed of value-range analysis, which captures the binding-time of upper bit-ranges, is an encouraging step in this direction. To capture the binding-time of lower bit-ranges, we might consider a factorization-based integer-arithmetic analysis.

References

- [1] William Blume, Rudolf Eigenmann, "Symbolic Range Propagation," *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, California, April 25-28 1995, pp. 357-363.
- [2] William H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Engineering*, vol. SE-3, no. 3, pp. 243-250, May 1977.
- [3] Luke Hornof, Jacques Noyé, "Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity," *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, Amsterdam, The Netherlands, June 12-13 1997, pp. 63-73.
- [4] Chunho Lee, Miodrag Potkanjak, William H. Mangione-Smith, "Media-Bench: a tool for evaluating and synthesizing multimedia and communication systems," *Proc. 30th International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, Dec. 1-3 1997, pp. 330-333.