# Range Analysis with Abstract Interpretation

Fall 2002 CS 263

Semester Project

Yury Markovskiy (13099223)
yurym@cs.berkeley.edu

December 18, 2002

## 1 Introduction

The knowledge of the precise range of variable values in a given program can be used in a variety of contexts, including compiler optimization, program checking and computing the minimum bit-width requirements. The last is gaining importance in the reconfigurable architecture community, where underlying substrates allow construction of customizable variable width data-paths. Empirical studies such as [2] demonstrate that many applications provide rich opportunities for bit-width specialization on reconfigurable logic. However, empirical studies require running a program with a wide range of input datasets in order to determine the minimum bit-width of variables.

The focus of this survey is to describe *static* analyses used to determine ranges of variables and consequently the number of bits required for their representation. While statically computing variable ranges precisely is often impossible in practice, the analyses we will examine employ abstract interpretation to approximate the variable ranges. The choice of abstract domain and the mapping between its members and the concrete values creates a trade-off between the quality of results and the analysis running time. Abstract interpretation [3], although imperfect, is a good fit to solve this problem due to the necessity to perform fixed-point computations. Both works presented in this survey found a way to overcome this challenge, but paid the penalty in quality.

This report will compare and contrast two efforts from MIT and CMU motivated by the problem of computing the minimum number of required bits for synthesis of code in a high-level language onto hardware. The first, *Bitwise* compiler [7] from MIT discussed in Section 2, performs variable range analysis to compute the most significant unused bits for a given variable in a particular program context. The second, BitValue algorithm developed at CMU [1], takes a different approach to solving this problem. The algorithm, which we describe in Section 3, attempts to compute the binding time and the value for each individual bit in a variable.

One common problem in both projects, that limits precision and attainable quality of results, is fixed-point computation (convergence or halting point in the analysis). In Section 4, we will describe the *symbolic* variable range analysis developed by Rugina *et al.* at MIT [6], which has one key advantage over the abstract interpretation. In this analysis, solving a linear programming optimization problem replaces fixed-point computation, allowing the author's implementation to obtain precise variable bounds in polynomial time. Neither of the projects discussed in this survey took advantage of this work on computing symbolic variable bounds, but resorted to simple heuristics to avoid accurate fixed-point computation (*e.g.* setting an upper bound on the number of iterations of the solver).

## 2 Value Range Analysis

William Harrison published one of the first works on using abstract interpretation for variable value range analysis [5]. He proposed the framework which combined two mechanisms *range propagation* and *range analysis* to avoid potentially very expensive fixed-point computation on looping constructs. This framework was implemented without any

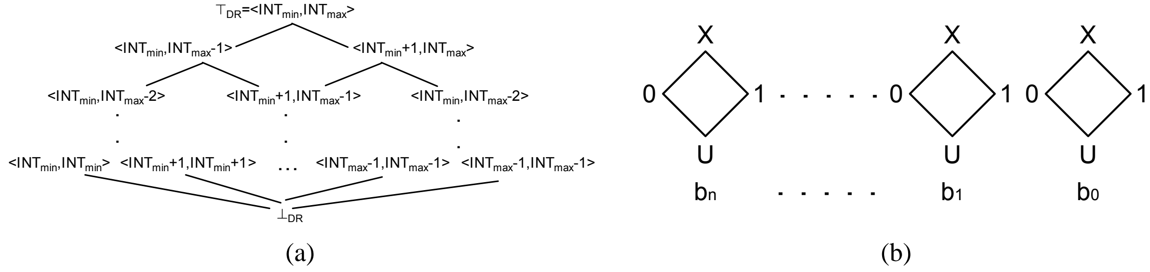


Figure 1: Examples of lattices used to represent range and bit values. *Bitwise* compiler uses lattice (a), and BitValue algorithm uses lattice (b).

fundamental modifications in both of the projects that will be compared in this survey. However, before diving into the implementation details, let us summarize these two key mechanisms.

*Range propagation* is a simple algorithm that uses the data and the conditional structure of a program to derive and propagate refinements in the accuracy of range information. For convenience and simplicity, the programs analyzed are first transformed into the Static Single Assignment (SSA) form. A range is defined for each point in the program, and the algorithm iteratively computes refinements of range information using specially defined transfer functions that operate on ranges. Unfortunately, range propagation process is not inductive in nature, and thus the presence of loops in the control flow of a program may severely limit its results. Thus, range propagation is effectively used to work on straight line code, requiring an alternative approach for loops.

*Range analysis* is an algorithm which tracks the changes to a variable at each point of the program, but does so in complete ignorance of the conditional structure of the loop. The information obtained is then used as a base for induction to derive a range of values for the variable. Ignoring conditional branches in the loop structure limits the accuracy of this technique for range analysis. However, if the problem is to identify unused or compile-time bound bits in a variable, this result with limited accuracy still may suffice.

The results obtained from *range propagation* and *range analysis* are intersected, while *range propagation* executes allowing the analysis to proceed with more accurate results. Harrison does not present implementation details or results for the combination of these techniques, but two projects discussed below do.

## 2.1 *Bitwise* compiler

The developers of the *Bitwise* compiler [7] closely followed the framework outlined by Harrison. Their primary concern is with integral data types such as `char`, `short`, and `int`, since these are the types that map directly to binary two's complement arithmetic in hardware. The designer chose an abstract domain of ranges, represented as pairs of integers $\langle l, m\rangle$, which are the smallest and the largest values respectively for members of the range. The ranges are partially ordered by inclusion as shown on the lattice on Figure 1a. The ordering allows a variety of required operations on the members of this abstract domain, each corresponding to a SSA node.

In order to perform range propagation, a number of operators have been defined on ranges. Of particular importance are data-range union ($\sqcup$) and intersection ($\sqcap$), which are defined intuitively as

$$\begin{aligned}\langle a_l, a_h\rangle \sqcup \langle b_l, b_h\rangle &= \langle min(a_l, b_l), max(a_h, b_h)\rangle\\ \langle a_l, a_h\rangle \sqcap \langle b_l, b_h\rangle &= \langle max(a_l, b_l), min(a_h, b_h)\rangle\end{aligned}$$

Using these basic operations, for each SSA node we define two transfer functions for forward and backward range propagation. The majority of these transfer functions are trivial to define, and we show several interesting ones on Figure 2. The reader is referred to [7] for a complete listing. In the *Bitwise* compiler the ranges are propagated in both directions, which significantly improves the precision of ranges computed by the algorithm.

The compiler takes advantage of additional information to seed its analysis and refine the ranges as they are computed. Structures such as arrays and bit-masks are particularly useful. The first provides a range for index variables,

$$
\begin{array}{llcl}
\text{(a)} & b_\downarrow = \langle b_l, b_h\rangle & & b_\uparrow = b_\downarrow \sqcap \langle a_l - c_h, a_h - c_l\rangle \\
 & c_\downarrow = \langle c_l, c_h\rangle & \downarrow & c_\uparrow = c_\downarrow \sqcap \langle a_l - b_h, a_h - b_l\rangle \\
 & & a = b + c & \\
 & a_\uparrow = \langle a_l, a_h\rangle & \downarrow & a_\downarrow = a_\uparrow \sqcap \langle b_l + c_h, b_h + b_l\rangle \\
\hline
\text{(b)} & x^b_\downarrow = \langle b_l, b_h\rangle & & x^b_\uparrow = x^b_\downarrow \sqcap x^a_\uparrow \\
 & x^c_\downarrow = \langle c_l, c_h\rangle & \downarrow & x^c_\uparrow = x^c_\downarrow \sqcap x^a_\uparrow \\
 & & x^a = \phi(x^b + x^c) & \\
 & x^a_\uparrow = \langle a_l, a_h\rangle & \downarrow & x^a_\downarrow = x^a_\uparrow \sqcap (x^b_\downarrow \sqcup x^c_\downarrow)
\end{array}
$$

Figure 2: A selected subset of transfer functions for forward and backward range propagation. The variables are subscripted with the direction in which they are computed. The operator in (a) adds two data ranges, and (b) merges the input ranges.

which may frequently refine a variable range from virtually unknown down to a small sequence of numbers. The approach, of course, has a serious problem, since if the original program uses erroneous array index range (*e.g.* in an out-of-bounds array access), an implementation that uses the refined range obtained by *Bitwise* will have a different behavior. In other words, the analysis implemented in this compiler assumes a program to be error-free. This is a serious problem, and common whenever a design is implemented in hardware, since a user may often specify the bit-width of the variable incorrectly producing the same result as would this compiler.

The example on Figure 3 illustrates in more detail the operation of the analysis described above. A range is defined for each variable of interest at each program point and initialized to $\top_{DR}$. Using Breadth-First Search, the compiler propagates ranges *forward* until either (1) a fixed-point is reached (*e.g.* a newly computed range does not refine an existing variable range), or (2) an opportunity exists to refine the range as shown below. The steps 1–6 of the analysis shown on Figure 3 are the results of forward range propagation. They are not particularly impressive, since the data-flow in the program snippet does not provide the compiler with sufficient information to further refine the results. However, the compiler takes advantage of `array` declaration, which although not shown on the figure, contains 10 elements. Beginning from step 7, where the range of `a5` is refined, the compiler *backward* propagates the ranges to obtain an impressively tight range of $\langle -2, 8\rangle$ in step 12 for variable `a0`. As was noted before, the compiler assumes that the program and its environment are error free, which here implies that `input()` subroutine will only return the values in the computed range. The analysis in the example terminates after step 12 because the fixed-point has been reached. However, in general case, the compiler would continue with forward propagation pass from the point of the most recent update (*i.e.* step 12) until the fixed point is reached. The analysis may iterate many times over the same code.

Although the range propagation often derives precise results on straight line code, its application is difficult on loops. Although traditional analyses simply iterate over the back edges in the control flow graph until the fixed-point is reached, this range propagation will saturate the ranges even in a simple loop-carried expression. This analysis does not take into account any static knowledge of loop bounds, and hence continuous iteration will produce the maximum range of its type for any induction variable. The designers employ a different strategy to classify mathematical sequences produced by induction variables and compute their closed-form solutions. We describe it next.

#### 2.1.1 Loops

*Bitwise* uses techniques from Gerlek *et al.* to classify induction variables by their sequence type and compute their closed form solutions [4]. Together with loop bounds, computed with the range propagation algorithm as explained above, a range for induction variables can be determined precisely. A combination of these two methods continuously

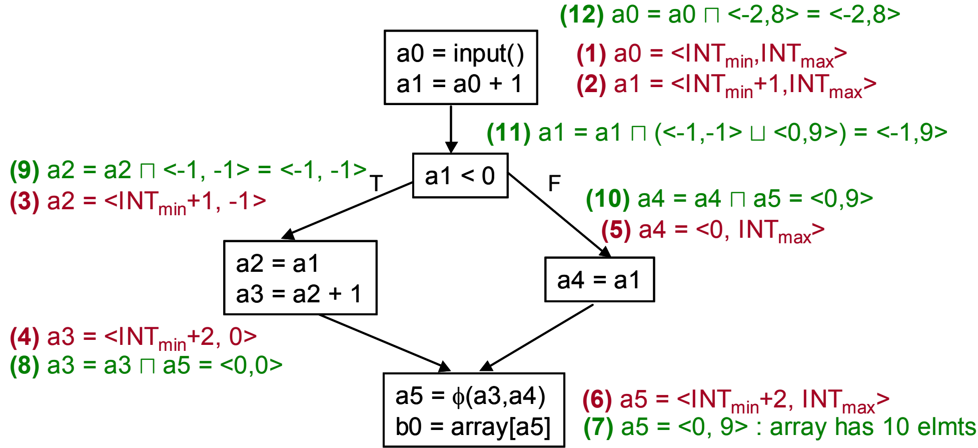


Figure 3: An example demonstrating the range propagation in *Bitwise* compiler. Steps 1–6 are the result of forward range propagation, and steps 7–12 result from backward propagation. No additional steps are needed because all range values have reached a fixed-point.

improve each other's results.

*Sequence Identification* is the first step to computing closed-form solutions for the induction variables. A *sequence* is a mutually dependent group of SSA instructions, a strongly connected component in the program's dependence graph. The algorithm presented by Gerlek *et al.* performs the following operations: (1) find all sequences in the loop, (2) topologically order all sequences according to any dependencies between them, (3) classify all sequence in order. The classification categories shown on Figure 4a are ordered by complexity, from `invariant`, the simplest and most precise, to `geometric` sequence, the most complex. Special abstract operators have been defined to allow the algorithm to classify the results of arithmetic on the induction variables. Figure 5 shows an example for $\oplus$.

Given a topologically sorted graph of sequences, each sequence is initially classified strictly by its dependencies on others. For example, on Figure 4b we show a simple program snippet with two sequences for variables `i` and `j`. Since the sequence with variable `i` is independent of any other and involves an addition to an invariant, it is automatically classified as *linear*. In turn, the sequence `j` depends directly on `i` and therefore must be classified as *polynomial* (in this case, quadratic).

Unfortunately, not all sequences are as easily identifiable as the one shown in the example, and hence are not properly classified. In this case, the algorithm simply iterates over the sequence until the fixed-point is reached. To decrease analysis running time, the number of iterations is bound by a *user-defined limit*, which leads to sub-optimal results. Fortunately, this is not a serious problem in most applications of interest (*multimedia*), since they use many boolean operations, shifts, and AND/OR-masks, that greatly reduce variable ranges.

Once all sequences have been classified, the algorithm proceeds to compute their *closed-form solutions*. Depending on the sequence class, *Bitwise* compiler invokes an appropriate solver. Detailed analysis and algorithms for these solvers are outside of scope of this survey. Their basic operation consists of traversing dependent nodes in each strongly connected component (sequence) and "adding up" the modifications to the induction variables. In a simple case of a *linear* induction variable, the solver aggregates all invariants added or subtracted from the variable of interest. In the example on Figure 4b, the solver produces closed-form expression $h + 1$ for `i` and $\frac{1}{2}h^2 + \frac{3}{2}h + 2$ for `j`, where $h$ is a "sequential loop counter," omitted for simplicity. Assuming that with help from the range propagation analysis, the upper bound of $h$ has been determined, it is now trivial to compute the precise range for `i` and `j`, and proceed with range propagation with this new refined result.

Neither the algorithm implemented in *Bitwise* compiler nor the publication by Gerlek describe the way to deal with situations when $\phi$ is one of the nodes in a sequence; *i.e.* branch inside the loop where varying control flow results in different modifications the value of the induction variable. At first glance, it seems that in many cases it will be

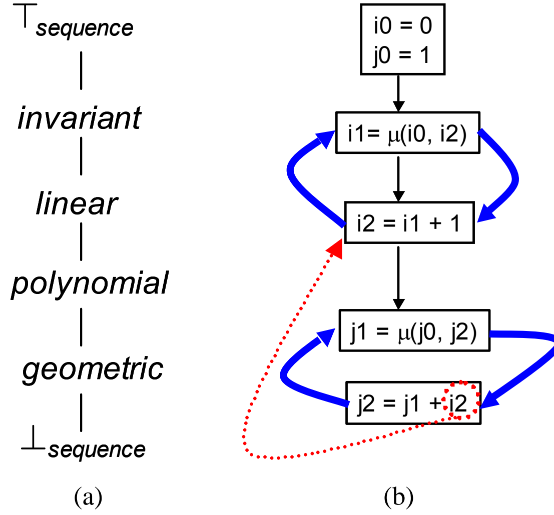


Figure 4: (a) The members sequence lattice are the categories for different types of induction variables. (b) An example of induction variable analysis.

| $\oplus$ | $\top$ | $inv$ | $lin$ | $poly$ | $geom$ | $\bot$ |
|---|---|---|---|---|---|---|
| $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $\bot$ |
| $inv$ | $\top$ | $inv$ | $lin$ | $poly$ | $geom$ | $\bot$ |
| $lin$ | $\top$ | $lin$ | $lin$ | $poly$ | $geom$ | $\bot$ |
| $poly$ | $\top$ | $poly$ | $poly$ | $poly$ | $geom$ | $\bot$ |
| $geom$ | $\top$ | $geom$ | $geom$ | $geom$ | $geom$ | $\bot$ |
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |

Figure 5: An example of abstract operator $\oplus$ used sequence classification of result of arithmetic expressions.

difficult to correctly classify an induction variable, and even more challenging to compute its closed form. However, in practice we can largely ignore the conditional structure of a loop body as was suggested by Harrison [5]. Since the range analysis attempts to compute the widest, all encompassing, "worst case" induction variable range, the analysis can be extended to widen the classification when unifying outputs from a $\phi$ node. The solver in this case must only analyze the sequence path, which gave the variable its classification, to derive the closed form solution. For example, assume that two paths exist in a loop. If only the first is taken, the variable would be classified as *linear*, but if the second is taken—*geometric*. For the purposes of range analysis, the variable can be classified as *geometric* and its closed form can serve to define its range based on a loop counter.

### 2.1.2 Results

*Bitwise* compiler developers present very impressive results for a wide range of benchmarks. Although it is true that all tested applications have a fine-grained, somewhat regular computational and control structure in common, and therefore are amenable to bit-width analysis, these are the applications that typically benefit the most from direct implementation in hardware. The compiler was tested with over fifteen applications and the number of eliminated bits was reported. The applications were also instrumented and executed to obtain the same numbers dynamically. On all but two applications, *Bitwise* compiler reports bit saving within 2–3% of the dynamic profile.

# 3 Bit-level Binding Time Analysis

An alternative approach to solving the problem of detecting and eliminating unused bits in variables is *binding time* analysis, which attempts to determine the time when the value of a bit is computed. Three such time categories have been defined: unknown, static (known at compile time), and dynamic (known at run time). A different way to define this analysis is to compute values for bits at compile time allowing marking all bits that change at run time as unknown.

The analysis procedure for this approach is very similar to range analysis. An abstract domain, forward and backward transfer functions are defined. Analysis proceeds until the fixed point is reached. Rather than describing these steps once more in detail, the discussion below will focus on differences between the approaches and the results.

## 3.1 BitValue algorithm

### 3.1.1 Straight line code

The way ranges are defined and used in the *Bitwise* compiler only permits elimination of the most significant bits in a word. The range propagation also is unable to compute any of the bits that remain *unchanged* during the program execution. Bit value analysis addresses these problems. Let us look at the implementation of BitValue algorithm from CMU [1]. BitValue defines a four valued lattice shown on Figure 1b. Each bit in a variable is assigned one of the values— `X`–*don't care*, `U`–*unknown*, `0` or `1`–if the value can be determined at compile time. A vector of 32 or 16 lattices, depending on the type, collectively represents the word. The advantage of this representation is clear. Any arbitrary bit can be classified as static (0 or 1) and optimized by the compiler. The BitValue algorithm is simply a bit-level version of constant folding and propagation widely employed in compiler optimizations.

The algorithm defines a collection of forward and backward bit-level operators on this abstract domain. Their definitions are mostly straightforward requiring the compiler to apply these transfer functions bit by bit in order (particularly important for add/subtract where carry must be propagated). The values obtained from the transfer functions corresponding to bit-wise OR, AND, XOR, and others generally produce precise results. Addition and subtraction also present no serious difficulties. However, integer division and multiplication usually saturate bit values to `X`, unknown. This problem is also present to a lesser degree in the range propagation, which uses the ranges shown on Figure 1a.

The chosen representation of bit values in this algorithm also frequently accounts for imprecise results, since the implementation must be conservative. For example, adding $\langle UUUU \rangle$ and $\langle UUUU \rangle$ must produce a vector with *five* $U$'s (dynamic values) to account for the carry. The subsequent operations on the result can only introduce even further imprecision. The representation used in the *Bitwise* compiler may have advantage over the bit value lattice vector in that it would be able to predict more accurately the range and thus dynamic bits in a given variable. However, neither publication presented any evidence that an alternative range or bit-value representation was quantitatively evaluated.

### 3.1.2 Loops

BitValue algorithm handles loops in the same way as the straight line code, by iteratively updating bit values until the fixed point is reached. Unfortunately, this approach frequently produces uninteresting results even when a simple loop carried dependence is present. For example, for a loop like `for (int i=0; i<2; i++)` the algorithm will infer a vector of bit values consisting of 32 $U$'s; *i.e.* all bits of `i` are dynamic. Clearly, only two bits are necessary to implement `i` for this piece of code.

To partially remedy the problem, the BitValue algorithm is executed together with a simplified version of the range analysis algorithm described earlier, which classifies induction variables identified in SSA graphs. The data from the range analysis is used to seed the bit value propagation algorithm, significantly refining the final results.

### 3.1.3 Results

The authors of BitValue algorithm also present impressive results obtained from a large variety of benchmarks. Unlike the case of *Bitwise*, however, they do not present a strong reference point (*e.g.* results obtained by dynamic profile),

which leaves the reader wondering how close the presented bit savings are from the optimum. Nevertheless, BitValue algorithm is able to determine that from 30 to 50% of variables in a given program require significantly fewer bits for representation than the size of the object of their type.

# 4 Symbolic Range Analysis

Although the approach discussed below performs range analysis without abstract interpretation, this survey would not be complete without addressing one of the key disadvantages of the two algorithms presented in previous sections: computing the fixed-point. Rugina and Rinard at MIT developed a framework for *symbolic* bounds analysis [6], which was used for range analysis on some of the same benchmarks tested by the developers of the *Bitwise* compiler. The results in terms of bits saved are very close to those reported by Stephenson *et al.* in [7]. The symbolic range analysis algorithm completely avoids the fixed point computation by turning the variable value range problem into a linear programming optimization.

Although a complete discussion of their work is outside the scope of this survey, the basic procedure is very straightforward. For each basic block, a set of *basic block input* variable range symbolic constraints are defined (one for each variable). The algorithm continues by computing a corresponding set of *basic block output* constraints on the range of each variable. Those are computed through data-flow analysis within a basic block in a similar fashion as range propagation pushes the modified ranges through SSA nodes. Finally, the algorithm sets up the linear programming optimization problem by defining the inequalities in such a way that *input* variable range in every basic block must include all of the *output* ranges in the predecessor basic blocks. If a basic block is part of the loop, it is easy to see that a range for an induction variable will be the *same* in all blocks forming this loop, which is intuitively true. Thus, solving the linear programming optimization is effectively a process of solving for the fixed point, without iteration over the SSA graph. The linear programming problem is defined to *minimize* the sum of sizes of all ranges (the size of range $\langle 4, 11\rangle$ is $8$), to obtain compact variable ranges.

As we mentioned previously, this technique produces comparable results to the *Bitwise* compiler, which utilizes abstract interpretation. For more detail on the algorithm for symbolic range propagation, refer to [6].

# 5 Conclusion

This survey presented two implementations of the analysis that employs abstract interpretation to compute unused bits that can be eliminated for efficient hardware synthesis. The *Bitwise* compiler performs range analysis using a combination of forward and backward range propagation techniques and loop range analysis, which classifies and computes closed-form solutions for induction variables. This systematic approach pays off through high quality results comparable to a dynamic profile. The BitValue algorithm approaches the same problem differently by offering similar analysis at the bit-level. While results are similarly impressive, their quality is difficult to evaluate without a clear reference point.

Lastly, we summarized a symbolic range analysis technique developed by Rugina and Rinard, which in addition to demonstrating results of similar quality to analysis with abstract interpretation, obviates the need for expensive fixed-point computation. One would hope that the symbolic range analysis can be incorporated in both aforementioned projects to further improve their results, especially in the code heavily dominated by control flow, where abstract interpretation may saturate the results.

# References

[1] Mihai Budiu, Majd Sakr, Kip Walker, and Seth C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. *Lecture Notes in Computer Science*, 1900:969–??, 2001.

[2] Eylon Caspi. Empirical study of opportunities for bit-level specialization in word-based programs. Master's thesis, UC Berkeley, 2000.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, USA., 1977.

[4] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[5] William H. Harrison. Compiler analysis of the value ranges for variables. In *IEEE Transactions on Software Engineering*, pages 243–250. IEEE Computer Society, May 1977.

[6] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[7] Mark William Stephenson. Bitwise: Optimizing bitwidths using data-range propagation. Master's thesis, MIT, 2000.