A Simple Value Range Propagation Pass for SUIF-1

Brian Greskamp
CS426, Fall 2003
greskamp@crhc.uiuc.edu

December 17, 2003

Abstract

Reconfigurable Computing (RC) uses reconfigurable logic devices such as FPGAs to assist general-purpose computation. In order to make efficient use of FPGA resources, it is essential to size datapaths and functional units to the minimal possible bitwidth that will ensure correctness. Since the high-level languages used to program RC systems do not require (or even allow) the user to manually specify datapath widths, the compiler must infer them. This paper presents a simple dataflow analysis for determining datapath widths in C programs and offers a preliminary SUIF-based implementation.

1 Introduction

Reconfigurable Computing (RC) is a developing field in computer architecture that aims to exploit programmable logic devices (*ie.* Field Programmable Gate Arrays) to accelerate general-purpose computation. The size of the RC device limits the number and type of computations that can be programmed into it. Fortunately, clever optimizations can increase the space-efficiency of computations in RC logic. *Bitwidth analysis* reduces the number of bits used to represent program variables. Not only are the number of storage (register) bits reduced, but the width of arithmetic operators also decreases. Narrow operators require fewer gates to implement and have fewer cascaded logic levels. Consequently, they are faster (lower propagation delay) and smaller than full-word operators. Reduced width operators also consume less power [4].

Although the RC machine represents a very different compilation target than a tradtional microprocessor, the system must present a familiar interface to the programmer. Specifically, it would be convenient to program RC machines in a familiar high-level language such as C. The programmer should not concern himself with the details of the hardware mapping, including the bitwidth of datapaths. Since few high-level languages require or even allow the user to specify datapath widths[1], the compiler must infer this information. Fortunately, some simple analysis can extract useful bitwidth information using clues such as array index and conditional expressions to restrict variable ranges and propagate those ranges throughout the program.

This paper presents one such simple method for bitwidth analysis, also known as "value range propagation". Section 2 begins the paper with some background and related work. A semi-formalized explanation of the chosen approach appears in section 3. Implementation-specific details for a SUIF-based compiler pass follow in section 4. Next, some simple tests in section 5 demonstrate the operation of the pass. Based on these results, suggestions for future work are given in section 6. Finally, section 7 concludes.

2 Background

Although the author's primary interest is hardware compilation, others have applied value range propagation to varied purposes. For example, Blume and Eigenmann [3] present a powerful symbolic range propagation

[1] The limited choice of `byte`, `half-word`, and `word` notwithstanding

method for dependence analysis in a parallelizing compiler, and static branch prediction motivates another presentation by Patterson [7]. In the hardware compilation domain, Ananian [1] was one of the first to apply bitwidth analysis — in a very literal sense. His scheme kept track of the number of bits needed to represent each quantity in the program, rather than the range of that quantity. The advantage, of course, is simplicity. A marked disadvantage is that arithmetic operations quickly cause bitwidths to grow. For example, adding two 12-bit numbers will produce a 13-bit result. Consequently, when implementing the *Bitwise* [8] analysis for their *DeepC* compiler, Stephenson *et. al.* chose to represent ranges instead of bitwidths. This paper draws heavily on their work.

The pass described in this paper is part of the *Amalgam* compiler project at UIUC. It's goals are similar to those of *DeepC*, with the exception that reconfigurable computing, rather than ASIC design is the target application. In reconfigurable computing, the kernels which are programmed into the reconfigurable logic are usually very small and they are typically leaves in the program callgraph. Therefore, the procedures that the *Amalgam* bitwidth pass must analyze are much less complex than those that *DeepC* may be required to process. Furthermore, at the current early stages of implementation, floating point computation is completely ignored in the *Amalgam* hardware compiler. However, Banerjee has produced interesting work on range propagation for floating point numbers with the goal of automatic fixed-point conversion [2].

3 Method

This section explains the dataflow framework that performs the range propagation. Unlike *Bitwise*, which operates on SSA form, this pass operates directly on the control flow graph. This is a dubious decision; in the author's opinion, the SSA approach is more elegant, but the limited time afforded this implementation precluded the possibility of writing an SSA-conversion pass. Ultimately, the *Amalgam* compiler will have a Gated Single Assignment [6] based backend, and it is likely that the bitwidth analysis will be ported to GSA as soon as development of the backend stabilizes. In the meantime, the CFG-based formulation serves to illustrate the major concepts

The analysis is bidirectional, comprising alternating phases of forward and backward propagation. Both phases operate on a control flow graph whose edges have been annotated with variable range information for each variable live on that edge. The set of ranges associated with an edge is called a *dictionary*. The forward and backward dataflow equations are solved by iteration, updating the dictionaries at each step. Iteration continues until a fixed point is found or a pre-set iteration limit is exceeded. Since the analysis is monotonic (*i.e.* ranges never increase across iterations), preempting the analysis at any time yields a correct but suboptimal result.

In this discussion, a "range" consists of either the value $undefined$ or an ordered pair $[min, max]$ where $min \leq max$. Any range with $min > max$ is equivalent to the value $undefined$. The communtative union ($\sqcup$) and intersection ($\sqcap$) operators defined on ranges have their intuitive meanings as clarified in Equation 1

$$\begin{gathered}
[a_l, a_h] \sqcap [b_l, b_h] = [max(a_l, b_l), min(a_h, b_h)] \\
R \sqcap undefined = R \\
[a_l, a_h] \sqcup [b_l, b_h] = [min(a_l, b_l), max(a_h, b_h)] \\
R \sqcup undefined = R
\end{gathered} \tag{1}$$

Each block b in the flowgraph has a forward transfer function f_b, which essentially interprets the instructions in that block, executing each one using interval arithmetic to determine the range of the left hand side. For single-exit blocks f_b operates on a dictionary from one of the block's in-edges and returns a new dictionary for the output. Conditional blocks have two transfer functions: $f_{b,true}$ returns the dictionary for the 'true' branch and $f_{b,false}$ handles the 'false' branch. For each block, backward transfer functions f_b^{-1} are also defined. The backward functions operate on dictionary values from an out-edge and return values for the in-edges. Transfer functions for each block are obtained by composing the transfer functions for each expression in the block. As examples of transfer functions for expressions, Equation 2 gives the forward transfer function for multiplication and Equation 3 gives the backward function for addition. Similar

functions are required for every C operator[2].

$$[a_l, a_h] \times [b_l, b_h] = [min(a_l b_l, a_h b_l, a_l b_h, a_h b_h), max(a_l b_l, a_h b_l, a_l b_h, a_h b_h)] \tag{2}$$

$$\begin{aligned} C &= A + B \\ A' &= A \sqcap [c_l - b_h, c_h - b_l] \\ B' &= B \sqcap [c_l - a_h, c_h - a_l] \end{aligned} \tag{3}$$

Propagation across array access expressions differs from the simple arithmetic examples given above. Any expression that serves as an array index has its range reduced to $[0, size - 1]$, where $size$ is the declared dimension of the array. The process also works with multidimensional arrays. Handling of conditional expressions is also special. For each conditional, the ranges for which the condition may be false are propagated down the false branch, while those where it may be true are propagated down the true branch.

For a single-exit block, Equation 4 and Equation 5 govern the dictionary updates for the forward and backward phases, respectively. $R(i, j)$ refers to the range dictionary associated with the edge $i \rightarrow j$. Equation 6 and Equation 7 provide the forward and backward equations for a single-entry conditional block. The most general case, the multiple-entry, two-exit block is governed in the forward case by Equation 8 and in the backward case by Equation 9, where $path(i, j)$ is the set of in-edges of block i from which execution can reach successor j. Computation of $path(i, j)$ occurs in the forward-propagation phase where the alorithm sees for each input dictionary whether the branch condition is always false, always true, or sometimes true. A major shortcoming of the current implementation is that the CFG is constructed such that multiple-entry multiple-exit nodes never occur. Section 4 discusses this in more detail.

$$R(i, succ(i)) \sqcap = \bigsqcup_{j \in pred(i)} f_i(R(i, j)) \tag{4}$$

$$\forall j \in pred(i) : R(j, i) \sqcap = f_i^{-1}(R(i, succ(i))) \tag{5}$$

$$\begin{aligned} R(i, true_succ(i)) \sqcap &= f_{i,true}(R(i, pred(i))) \\ R(i, false_succ(i)) \sqcap &= f_{i,false}(R(i, pred(i))) \end{aligned} \tag{6}$$

$$R(pred(i), i) \sqcap = R(i, true_succ(i)) \sqcup R(i, false_succ(i)) \tag{7}$$

$$\begin{aligned} R(i, true_succ(i)) \sqcap &= \bigsqcup_{j \in pred(i)} f_{i,true}(R(j, i)) \\ R(i, false_succ(i)) \sqcap &= \bigsqcup_{j \in pred(i)} f_{i,false}(R(j, i)) \end{aligned} \tag{8}$$

$$\begin{aligned} \forall j \in path(i, true_succ(i)) : R(j, i) \sqcap &= R(i, true_succ(i)) \\ \forall j \in path(i, false_succ(i)) : R(j, i) \sqcap &= R(i, false_succ(i)) \end{aligned} \tag{9}$$

An alternative method of forward propagation involves examining the dictionaries for all of the in-edges at a node and taking their union. The result of the union operation can then be propagated through the block. This is, however, less accurate in some cases. Consider the example in Figure 1. If the union is taken before applying the transfer function, `d` is not determined to be constant, whereas the scheme formalized above determines that $d = 3$. However, even in the stronger scheme, if another block is inserted immediately before the one containing the assignment to `d`, then the analysis fails to determine that d is constant. Whether the two approaches are substantially different in real programs remains to be seen.

[2]The backward propagation functions are as yet nowhere near complete

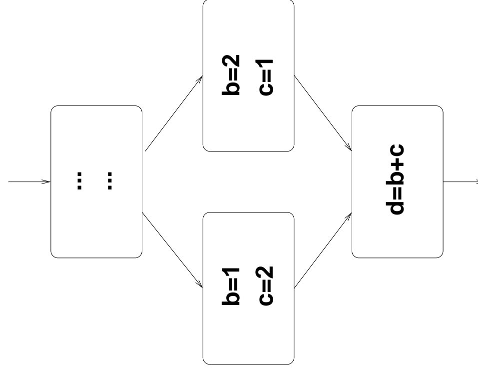

Figure 1: A tricky constant propagation problem.

4 Implementation

The bitwidth analysis pass has been implemented in the SUIF-1 compiler infrastructure. It operates on a high-level intermediate representation where array index and conditional expressions remain explicit. In a lower-level IR, array accesses would be dismantled into a series of address computations and memory loads, but as explained in the previous section, the pass needs access to the full array index expressions. While simplifying bitwidth analysis, using high-level IR complicates subsequent passes in the optimization chain, since they must all take account of the range information. For example, a common subexpression elimination pass applied at a lower level would have to re-calculate range values for each subexpression it moved. Theses other passes would not need to be range-aware if the bitwidth analysis pass were last in the optimization chain, but this would be extremely difficult to implement.

The `bitwidth` pass begins by constructing a minimal control flow graph from the SUIF IR. Each node in the graph is a single SUIF-IR statement. It then applies live variable analysis to determine the live-out variables for each block and initializes dictionary entries for each variable. When initializing the dictionaries, the pass sets all variables to their full declared width. In the case that the user has explicitly restricted a variable range with the `range` pragma, the restricted range is used.

Iteration of the forward and backward propagation passes then proceeds as follows: `prop_forward_proc` iterates over all basic blocks performing forward propagation. For each conditional node that is encountered, a test is performed to determine whether the 'true' and 'false' branches are both reachable. If either is determined to not be reachable, it is marked for later deletion. `prop_backward_proc` then iterates over all nodes in a similar fashion performing back-propagation. Next, if any unreachable nodes exist, they are removed from the CFG and all of the dictionaries are re-initialized to their default values. The iteration of forward and backward phases continues until a fixed point is found or the iteration limit is exceeded. Finally, the program dumps the contents of each dictionary to `stdout` in a human-readable form, since the question of how to communicate the output of the pass to the rest of the *Amalgam* toolchain is still unresolved.

In order to use the pass, one must first convert the C language program into a suitable SUIF file. The `scc` compiler from the SUIF distribution converts the C file to a high-level SUIF file. Next, the SUIF oprimizer, `porky`, is used to dismantle loops and scoping blocks. The result of the `porky` run is a `.spx` file suitable for input to the bitwidth pass. The bitwidth pass is an executable called `bitwidth` that takes as single argument the name of the `.spx` file to process. Output appears on stdout. Unfortunately, the output is not very easy to decipher.

```
void test(int a, int b)
{
        int c;
        #pragma range a [0,20]
        #pragma range b [-20,5]

        c=a*b;
        /* c=[-400,100] */
        c/=(b+4);
        /* c=[-400,400] */
        a--;
        /* a=[0,19] */
}
```

Figure 2: Forward propagation test.

```
void test(int a) {
        int array[100];
        int b;
        int c;

        c=a+2;
        /* c=[2,101], a=[0,99] */
        b=array[a];
        /* b=unlimited, a=[0,99] */
}
```

Figure 3: Backward propagation test.

5 Results

This section presents some preliminary results obtained from running the pass on simple code fragments. The example in Figure 6 demonstrates the use of pragmas to restrict variable ranges and shows how forward propagation carries these values through arithmetic operations. The comments reflect the result of executing their preceeding statements. Backward propagation is illustrated in Figure 3. The array access clamps the range of `a` to $[0, 99]$ because `array` contains 100 elements. The reduced range for `a` is then propagated backward to the preceeding statements, reducing the range of `c`.

Figure 4 demonstrates how branches that are never taken are removed from the control flow graph so that better range limits can be calculated. Since `a` is always greater than 80, the assignment `c=-1000` can never occur, so the final range of `c` does not include 1000. Figure **??** shows how ranges propagate through conditional statements in which both the 'true' and 'false' branches are reachable. Variable `b` receives disjoint values in the `if` and `else` parts of the conditional.

Finally, the limited loop-handling capabilities of the current pass are shown in **??**. Although the ranges for the array index variable are correct inside and outside of the loop, `sum` remains at its full width. The following section suggests an improved analysis which would allow range bounds to be established for `sum`.

6 Future Work

Treatment of loops in the current implementation is extremely poor. Calculating accurate bounds for loop-carried variables may require up to 2^{32} executions of the loop body. Obviously, this is impractical. It is preferrable to analyze the loop symbolically, solving for the values of all loop-carried variables in terms of the loop indices. Ranges for the loop carried values can then be trivially calculated from the loop index ranges. Wolf *et. al.* give algorithms for solving a variety of common loop types, including those with linear, periodic, polynomial, and geometric sequence variables [5].

```
void test(int a, int b)
{
        int c, d;
        #pragma range a [0,100]
        #pragma range b [-50,10]

        /* a=[0,100], b=[-50,10] */
        c=50;
        /* c=[50,50] */
        if(a>80) {
                /* a=[81,100] */
                c=a+b;
                /* c=[31,110] */
        }
        else if(a+b>100) {
                /* DEAD CODE */
                c=-1000;
        }
        /* c=[31,110] */
        d=c;
        /* d=[31,110] */
}
```

Figure 4: Deadcode elimination test.

```
void test(int b)
{
        int c=10;
        int d;
        #pragma range b [0,1000]

        if(b<100) {
                /* b=[0,100] */
                c=b;
                /* c=[0,99] */
        }
        else {
                /* b=[100,1000] */
                c-=b;
                /* c=[-990,-90] */
        }
        /* c=[-990,99] */
        d=c;
        /* d=[-990,99] */
}
```

Figure 5: Conditional propagation test.

```
void test(void)
{
        int array[100];
        int i;
        int sum;
        int final;
        #pragma range array [0,1000]

        sum=0;
        /* sum=[0,0] */
        for(i=0;i<50;i++) {
                /* i=[0,49], sum=unlimited */
                sum+=array[i];
                /* sum = unlimited */
        }
        /* i=[50,50] */
        final=i;
        /* final=[50,50] */
}
```

Figure 6: Loop test.

In order to incorporate the improved loop solver, the pass could be modified as follows. Instead of immediately dismantling all loops into tests and jumps, they could initially remain intact in the high-level IR. Invoking the loop solver on all loop bodies would provide closed-form solutions for some of the loops. Next, in the CFG construction phase, all loop bodies for which a closed form solution was determined would be collapsed to single nodes having the transfer function computed by the loop solver. All unsolved loops would be dismantled and added to the CFG as in the current scheme. With the CFG contstructed, the iterative dataflow solver could proceed as before iterating to a fixed point. Finally, the collapsed loops would be expanded in the CFG, using the predetermined transfer function to assign ranges to each loop-carried variable based on the (now known) ranges of the loop's live-in and index variables.

A simpler problem exists with the way the CFG is constructed. Currently, the CFG comprises minimal nodes, containing single SUIF statements. One consequence of this is that jump targets (*i.e.* `label` instructions) occupy their own block. Therefore, the only multiple-entry blocks are `label` statements! This causes suboptimal in situations such as Figure 1. The problem is easily solved by modifying the propagation routines to operate on maximal basic blocks. Whether or not the change will significantly improve performance is unknown.

7 Conclusions

References

[1] C. S. Ananian. The static single information form. Master's thesis, Massachussets Institute of Technology, 1999.

[2] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe. Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware designs. In *Proc. of the 2003 Conference on Field Programmable Custom Computing Machines*, 2003.

[3] W. Blume and R. Eigenmann. Demand driven, symbolic range propagation. In *Proc. of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, 1995.

[4] Y. Cao and H. Yasuura. Leakage power reduction using bitwidth optimization. In *Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI2002)*, July 2002.

[5] M. P. Gerlek, E. Stoltz, and M. 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.

[6] P. Havlak. Construction of thinned gated single-assignment form. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 477–499, Portland, Ore., 1993. Berlin: Springer Verlag.

[7] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

[8] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *SIGPLAN Programming Language Design and Implementation*, 2000.