BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations

Mihai Budiu   Seth Copen Goldstein

June 2000

CMU-CS-00-141

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An abridged version of this text appeared in the Proceedings of 6th International Euro-Par Conference, August 2000, published in LNCS 1900 by Springer Verlag.

Abstract

We present a compiler algorithm called BitValue, which can discover unused and constant bits in dusty-deck C programs. BitValue uses forward and backward dataflow analyses, generalizing constant-folding and dead-code detection at the bit-level. This algorithm enables compiler optimizations targeting special processor architectures for computing on non-standard bitwidths.
Using this algorithm we show that up to 36% of the computed bytes are thrown away; also, we show that on average 26.8% of the values computed require 16 bits or less (for programs from SpecINT95 and Mediabench). A compiler for reconfigurable hardware uses this algorithm to achieve substantial reductions (up to 20-fold) in the size of the synthesized circuits.

This work was supported by DARPA contract DABT63-96-C-0083 and an NSF CAREER grant.

Keywords: Compilation, dataflow analysis, reconfigurable hardware, CAD tools

1 Introduction

As the natural word width of processors increases, so grows the gap between the number of bits used and those actually required for a computation. Recent architectural proposals have addressed this inefficiency by providing collections of narrow functional units or the ability to construct functional units on the fly. For example, instruction set extensions which support subword parallelism (e.g. [10]), Application-Specific Instruction-set Processors (ASIPs) (e.g. [9]), and reconfigurable devices (e.g. [11]) all allow operations on operands which are smaller than the natural word size.

Reconfigurable computing devices are the most efficient at supporting arbitrary size data because they can be programmed post-fabrication to implement functions directly as hardware circuits. In such devices it is possible to create functional units which exactly match the bit-widths of the data values on which they compute.

State of the art methods for using the special architectural features require the programmer to use macro libraries or specify the bit-widths manually, a tedious and error-prone process. Furthermore, there is little or no support in high-level languages for specifying arbitrary bit-widths.

In this paper we present BitValue, an algorithm which enables the compilation of unannotated high-level languages to take advantage of variable size functional units. Our technique uses dataflow analysis to discover bits which are independent of the program inputs (constant bits) and bits which do not influence the program output (unused bits). By eliminating computations of both constant and unused bits the resulting program can be made more efficient.

BitValue generalizes constant folding and dead-code elimination to operate on individual bits. When used on C programs, BitValue determines that a significant number of the bit operations performed are unnecessary: on average 14% of the computed bytes in programs from SpecINT95 and 21% of the bytes in Mediabench are useless. Our technique also enables the programmer to use standard language constructs to pass width information to the compiler using masking operations.

Narrow width information can be used to help create code for sub-word parallel functional units. It can also be used to automatically find configurations for reconfigurable devices. BitValue has been implemented in a compiler which generates configurations for reconfigurable devices, reducing circuit size by factors of three to twenty.

Contributions. We summarize here the contributions of this paper:

- We formulate the problem of bit-value inference as a dataflow problem, of inferring the value of each computed bit (as one of "constant", "useful bit", "don't care").
- We give an algorithm to solve the bit-value inference problem.
- We evaluate the implementation of our algorithm in a C compiler and in a compiler for reconfigurable hardware.
- We measure the effects of our analysis for detecting narrow widths in programs from SpecINT95 and Mediabench.

- We measure the reductions in circuit size due to our algorithm in a compiler for reconfigurable hardware.

In Section 2 we present our BitValue inference algorithm. Section Section 3 shows the algorithm in action on two examples. Results for the implementation in a C compiler are in Section 4 and for a reconfigurable hardware compiler in Section 5. Related work is presented in Section 6 and we conclude in Section 7.

2 The BitValue Inference Algorithm

For each bit of an arbitrary-precision integer, our algorithm determines whether (1) it has a constant value, or (2) its value does not influence the visible outputs of the program. Those two possibilities are similar to constant folding and dead code elimination, respectively. In our setting, however, these are performed at the bit-level within each word.

We can cast our problem as a *type-inference* problem, where the type of a variable describes the possible value that each bit can have during an execution of the program. The BitValue algorithm solves this problem using dataflow analysis. In this section we introduce first the dataflow lattice, we present the transfer functions, we give an outline of the algorithm and conclude with two examples.

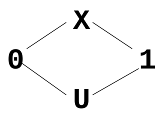

Figure 1: *The bit values lattice. The ordering is defined by the "information content".*

The Bit-value Lattice. We represent the bit values by one of: $\langle \texttt{0} \rangle$, $\langle \texttt{1} \rangle$, *don't know* (denoted by $\langle \texttt{u} \rangle$) and *don't care*, (denoted by $\langle \texttt{x} \rangle$). Let us call this set of values $\mathcal{B}$. Some bits are constant, independent of the inputs and control flow of the program; such bits are labeled with their value, $\langle \texttt{0} \rangle$ or $\langle \texttt{1} \rangle$. A bit is labeled $\langle \texttt{x} \rangle$ if it does not affect the output; otherwise a bit is labeled $\langle \texttt{u} \rangle$. These bit values form a lattice, depicted in Figure 1. We write $\cup$ and $\cap$ for sup and inf in the lattice respectively. The top element of the lattice $\top$ is $\langle \texttt{x} \rangle$ and the bottom $\perp$ is $\langle \texttt{u} \rangle$.

The Bit String Lattice. We represent the type of each value in the program as a string of bits. We write $\mathcal{B}^*$ to denote all strings of values in $\mathcal{B}$. For example, for the C statement[1] `unsigned char a = b & 0xf0`, we determine that the type of `a` is $\langle \texttt{uuuu0000} \rangle$, and that the type of `b`, assuming it is dead after this statement, is $\langle \texttt{uuuuxxxx} \rangle$. A regular 8-bit value about which we know nothing is represented as $\langle \texttt{uuuuuuuu} \rangle$. We write the bitstrings like numbers, with the most significant bit to the left. $\perp$ is an infinite string of $\langle \texttt{u} \rangle$s, and $\top$ is the empty string.

[1]ANSI C doesn't mandate the size of a `char` or `int`; we just exemplify in the context of a plausible implementation.

The bitstrings also form a lattice $\mathcal{L}$. The $\cup$ and $\cap$ operations in $\mathcal{L}$ are performed bitwise (i.e. $\langle\texttt{ab}\rangle \cup \langle\texttt{cd}\rangle = (\langle\texttt{a}\rangle \cup \langle\texttt{c}\rangle)(\langle\texttt{b}\rangle \cup \langle\texttt{d}\rangle))$, where we have used juxtaposition to denote concatenation. For example, $\langle\texttt{xu}\rangle \cup \langle\texttt{0x}\rangle = \langle\texttt{xx}\rangle$ and $\langle\texttt{xu}\rangle \cap \langle\texttt{0x}\rangle = \langle\texttt{0u}\rangle$.

When applied to strings of different lengths, $\cup$ gives a result of the shorter length, while $\cap$ gives a result of the bigger length. The shorter value is sign-extended in the lattice for the $\cap$ computation: a string representing an unsigned number is sign-extended with $\langle\texttt{0}\rangle$ bits, while a string representing a signed number is sign-extended with its most significant bit. For example, for signed numbers, $\langle\texttt{1u}\rangle \cap \langle\texttt{u0x}\rangle = \langle\texttt{11u}\rangle \cap \langle\texttt{u0x}\rangle = \langle\texttt{uuu}\rangle$, while $\langle\texttt{1u}\rangle \cup \langle\texttt{u0x}\rangle = \langle\texttt{1u}\rangle \cup \langle\texttt{0x}\rangle = \langle\texttt{xx}\rangle$.

The Transfer Functions. To carry the dataflow analysis we need to show how each operation in the program computes on values in the lattice $\mathcal{L}$. We thus need to give the definition of the *transfer functions* for these operations.

The forward transfer function propagates constant bits forward through the program. The backward transfer function propagates *don't care* bits from destinations to sources. We associate to each operation in the program one forward and one backward transfer function which indicate how the operation computes on strings in $\mathcal{L}$.

For example, for the "and" operation, denoted in C by `&`, we can completely describe the forward transfer function by specifying how it operates for strings of only one bit. To compute on longer strings we apply it bitwise. Table 1 gives the definition for individual bits. To apply the "and" function to arbitrary strings in $\mathcal{L}$, the shorter string is sign-extended to the length of the longer one before we apply the operation bitwise.

&	$\langle\texttt{x}\rangle$	$\langle\texttt{0}\rangle$	$\langle\texttt{1}\rangle$	$\langle\texttt{u}\rangle$
$\langle\texttt{x}\rangle$	$\langle\texttt{x}\rangle$			
$\langle\texttt{0}\rangle$		$\langle\texttt{0}\rangle$	$\langle\texttt{0}\rangle$	$\langle\texttt{0}\rangle$
$\langle\texttt{1}\rangle$		$\langle\texttt{0}\rangle$	$\langle\texttt{1}\rangle$	$\langle\texttt{u}\rangle$
$\langle\texttt{u}\rangle$		$\langle\texttt{0}\rangle$	$\langle\texttt{u}\rangle$	$\langle\texttt{u}\rangle$

Table 1: *The transfer function for the "and" C function for strings of one bit. The empty slots indicate cases which can never arise.*

This definition is quite intuitive: we can apply the function bitwise, because this is how the real "and" function operates: the i-th bit in the input influences only the i-th bit in the output. For constant values the transfer function has to operate like the real function. For $\langle\texttt{u}\rangle$ values it has to assume the worst-case value for the bit: it can be either 0 or 1, and the result is the worst ($\cap$) of these two cases.

The backward transfer function for the "and" operation also operates bitwise. For `a & b = c`, the backward transfer function tells us the values of `a` and `b` in $\mathcal{L}$ given the value of `c`. (Actually we will generalize the backward transfer function to also depend on the known input bits, i.e. if we know that a bit of `a` is $\langle\texttt{0}\rangle$, the corresponding bit of `b` is $\langle\texttt{x}\rangle$.)

Table 2 shows that a don't care in the output "propagates" to both inputs as a don't care, as we would expect (because "and" is symmetric in its inputs we display only the dependence from the output to one input).

Output	⟨x⟩	⟨0⟩	⟨1⟩	⟨u⟩
Input	⟨x⟩	⟨u⟩	⟨u⟩	⟨u⟩

Table 2: *A reverse transfer function for the "and" C function for strings of one bit.*

In order to certify the correctness of our algorithm, we need to prove that the transfer functions we define are monotone and conservative. We define $\mathcal{A} : (\mathbb{N} \to \mathbb{N}) \times \mathcal{L} \to \mathcal{L}$, the forward transfer function of an operator in three steps. Given a unary[2] operation $f : \mathbb{N} \to \mathbb{N}$, $\mathcal{A}(f, \cdot)$ is its associated forward transfer function in $\mathcal{L} \to \mathcal{L}$.

A bitstring whose all bits have constant values denotes a single integer value. For such a bitstring the transfer function should behave identically to the corresponding function in the program: e.g. $\mathcal{A}(f, v) = f(v)$ if $v \in \{0, 1\}^*$. (To simplify the notation, if v is a bitstring with constant bits only (i.e. $v \in \{0, 1\}^*$), we denote with v both the bitstring v and the integer number represented by this bitstring. Notice that the bitstring ⟨11⟩ may represent either the number 3 or -1, depending on whether it is signed or unsigned. We assume that the "signedness" of a bitstring is carried together with the string.)

If a bitstring contains ⟨u⟩ bits, it no longer represents a single constant value, but a set of values. For example, ⟨u0⟩ represents the values ⟨00⟩ and ⟨10⟩. To capture this information we define an "expansion" function $\exp_u : \mathcal{L} \to 2^{\mathcal{L}}$, which takes a bitstring s and generates a set of bistrings: all bitstrings that can be obtained from s by replacing the ⟨u⟩ bits in s with some constant. For example $\exp_u(\langle\texttt{0u1u}\rangle) = \{\langle\texttt{0010}\rangle, \langle\texttt{0011}\rangle, \langle\texttt{0100}\rangle, \langle\texttt{0111}\rangle\}$. Notice that the expansion function is defined for strings which contain ⟨x⟩s too.

The transfer function $\mathcal{A}(f, \cdot)$ is conservative if $\forall s \in \{0, 1, u\}^*.\forall v \in \exp_u(s).f(v) \in \exp_u(\mathcal{A}(f, s))$; i.e. if v is in the set represented by s, then $f(v)$ must be in the set represented by $\mathcal{A}(f, s)$. We define thus $\mathcal{A}(f, s) = \bigcap_{y \in \exp_u(s)} f(y)$ for $s \in \{0, 1, u\}^*$.

To deal with don't care bits, we will make a similar argument. Each string which contains ⟨x⟩ bits actually represents a set of possible strings, in the same way: ⟨0x0⟩ can stand for either of ⟨000⟩ or ⟨010⟩. We define similarly $\exp_x(s) : \mathcal{L} \to 2^{\mathcal{L}}$ as the set of all bitstrings obtained from s by substituting the ⟨x⟩ bits will all possible constant values.

A bit is don't care if its value doesn't matter for the result. So we can choose any constant value for these bits to compute the result. We can define $\mathcal{A}(f, s) = \bigcup_{y \in \exp_x(s)} f(y)$. Then, for any $y \in \exp_x(s)$ we will have $f(y) \in \exp_x(\mathcal{A}(f, s))$.

Finally,

$$\mathcal{A}(f, v) = \bigcup_{y \in \exp_x(v)} \bigcap_{x \in \exp_u(y)} f(x).$$

The intuition behind this equation is the following: when we compute the transfer function in $\mathcal{L}$ for an input value, we can choose the most convenient values for the input bits which are marked ⟨x⟩, but we must "cover" with the result the entire space of possibilities for the bits marked ⟨u⟩. This definition can be easily extended to deal with n-ary operators.

For example, here is what the above definition yields for the C complementation ~

[2] These definitions are easily generalized for functions with multiple arguments.

operator when applied to $\langle \texttt{u0x} \rangle$:

$$\begin{aligned}
\mathcal{A}(\tilde{}, \langle \texttt{u0x} \rangle) &= \mathcal{A}(\tilde{}, \langle \texttt{u00} \rangle) \cup \mathcal{A}(\tilde{}, \langle \texttt{u01} \rangle) \\
&= (\mathcal{A}(\tilde{}, \langle \texttt{000} \rangle) \cap \mathcal{A}(\tilde{}, \langle \texttt{100} \rangle)) \cup (\mathcal{A}(\tilde{}, \langle \texttt{001} \rangle) \cap \mathcal{A}(\tilde{}, \langle \texttt{101} \rangle)) \\
&= ((\tilde{}\langle \texttt{000} \rangle \cap \tilde{}\langle \texttt{100} \rangle) \cup (\tilde{}\langle \texttt{001} \rangle \cap \tilde{}\langle \texttt{101} \rangle)) \\
&= ((\langle \texttt{111} \rangle \cap \langle \texttt{011} \rangle) \cup (\langle \texttt{110} \rangle \cap \langle \texttt{010} \rangle)) \\
&= \langle \texttt{u11} \rangle \cup \langle \texttt{u10} \rangle \\
&= \langle \texttt{u1x} \rangle
\end{aligned}$$

In practice we implement transfer functions which are simpler to compute (the expansion functions $\exp_u$ and $\exp_x$ can have as result a set with a number of elements exponential in the size of their argument). However, all our transfer functions are conservative approximations (in the function lattice $\mathcal{L} \to \mathcal{L}$) of the functions given by $\mathcal{A}(f, \cdot)$.

The backward transfer function will discover *don't care* bits in the input starting from the *don't care*s in the output. We define the backward functions using techniques from Boolean function minimization [6].

The notion of *don't care* input for a Boolean function f of n Boolean variables is well known: an input x_i is a *don't care* if the derivative of f with respect to x_i is zero: $\frac{\partial f}{\partial x_i} = 0$, i.e. $f|_{x_i=0} = f|_{x_i=1}$. We can view an operator which computes many bits (like addition) as a vector of Boolean functions, each computing one bit of the result. Let us denote with $\overline{x} = (x_n, x_{n-1}, \ldots, x_0)$ the input bits and with $\overline{y} = (y_m, y_{m-1}, \ldots, y_0)$ the output bits. If $f(\overline{x}) = \overline{y}$ we can say that $y_k = f_k(\overline{x})$, i.e. f_k is the function which computes the k-th bit of the output.

An input bit is don't care for an operator if it is a *don't care* for *all* the boolean functions f_i. We define the reverse transfer function for each f_k for each input bit x_i like this:

$$x_{i,k} = \begin{cases} \langle \texttt{x} \rangle \text{ if } y_k = \langle \texttt{x} \rangle \\ \langle \texttt{x} \rangle \text{ if } y_k \neq \langle \texttt{x} \rangle \text{ and } \frac{\partial f_k}{\partial x_i} = 0 \\ \langle \texttt{u} \rangle \text{ otherwise} \end{cases}$$

We can then compute the i-th input bit from all the k values like this: $x_i = \cap_k x_{i,k}$.

When some of the input bits have constant values, we consider the restriction of each f_k to the constant inputs when computing the don't cares. For instance, if $x_0 = \langle \texttt{0} \rangle$, then in the above formula we use $f_k|_{x_0=0}$ instead of f_k.

For example, let us see how the backward propagation operates for the "xor" operator, on the statement `c = a^b` when we know that the types of `a`, `b` and `c` are respectively $\langle \texttt{u0} \rangle$, $\langle \texttt{uu} \rangle$ and $\langle \texttt{xu} \rangle$; we expect the don't care bit of `c` (the most significant) to be propagated to `a` and `b`. The two bits of `c` are computed by two boolean functions, each having 4 inputs: $c_0 = f_0(a_0 = 0, a_1, b_0, b_1) = a_0\hat{\ }b_0$ and $c_1 = f_1(a_0, a_1, b_0, b_1) = a_1\hat{\ }b_1$. Because $c_1 = \langle \texttt{x} \rangle$, all the input bits of f_1 are $\langle \texttt{x} \rangle$.

Looking at the don't cares of f_0 we obtain that a_1 is a *don't care*, because $f_0|_{a_1=0} = f_0|_{a_1=1}$; b_1 is also a don't care of f_0. To summarize, the inputs of f_1 are $\langle \texttt{xx} \rangle$ and $\langle \texttt{xx} \rangle$. The inputs of f_0 are $\langle \texttt{ux} \rangle$ and $\langle \texttt{ux} \rangle$. The inputs of the "xor" will be computed taking the infimum of these values: $a = \langle \texttt{xx} \rangle \cap \langle \texttt{ux} \rangle = \langle \texttt{ux} \rangle$, and the same computation for b.

```
unsigned char
f(unsigned char c,
  unsigned char a)
{
    unsigned char d;
    d = (c + a) & 0x33;
    return (d >> 4)
           + (d << 2);
}
```

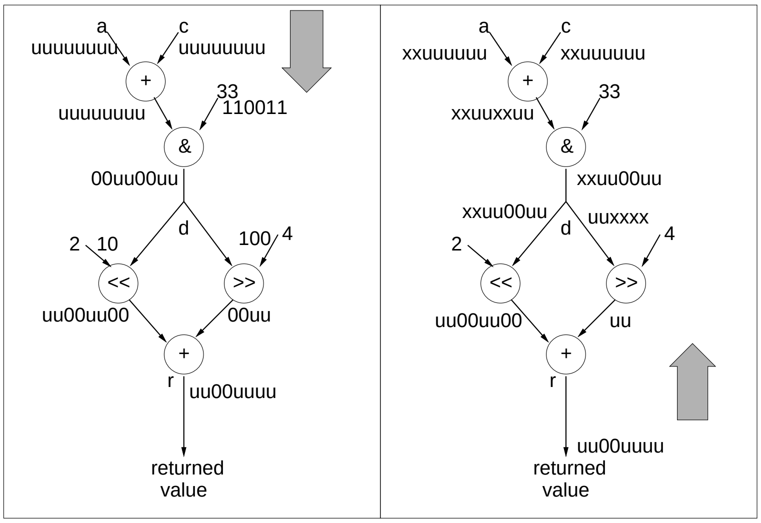

Figure 2: *A C function and the associated data-flow graph. The types inferred by forward (backward) propagation are shown in the left (right) figure. We assume that a* `char` *has 8 bits.*

When the algorithm described in Appendix A concludes the computation, each value will get a type combining the information from the forward and backward passes using a "sup". The final types will be $a = \langle \mathtt{u0} \rangle \cup \langle \mathtt{ux} \rangle = \langle \mathtt{u0} \rangle$ and $b = \langle \mathtt{uu} \rangle \cup \langle \mathtt{ux} \rangle = \langle \mathtt{ux} \rangle$.

In this example the fact that $a_0 = 0$ was not useful to infer more information in the backward propagation, but if we change the operator from `^` to `&`, this information provides the type $\langle \mathtt{x} \rangle$ for b_0.

In practice the transfer functions as given by the above definitions can be expensive to compute, so we resort to using monotone conservative approximations. Appendix C shows the current implementation we have for the various transfer functions.

The Dataflow Analysis. We compute the types using iterative dataflow analysis. We maintain for each value two types: the *best* type and the *current* type. The *best* type is initialized conservatively to $\perp$ and moves up in the lattice after each pass. The analysis alternates forward and backward dataflow passes, terminating when the *best* type does not change during a pass.

Each pass starts by initializing the *current* type for all the values $\top$, and proceeds to do the dataflow computation; during this computation the *current* types move down in the lattice until a fixed point is reached. At the end of each pass we update the *best* type: *best* $=$ *best* $\cup$ *current*.

Appendix A presents the complete pseudocode of the BitValue dataflow algorithm.

3 Examples

In this section we present two examples of the algorithm in action on two small programs.

```
char
f(unsigned a)
{
   unsigned short i, r=0;
   for (i=0; i < a; i++)
       r += i;
   return r;
}
```

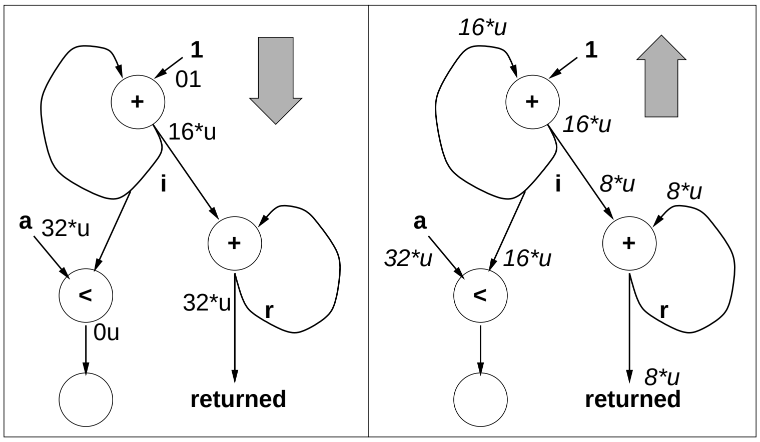

Figure 3: *An example with a loop and the associated data-flow graph. The types inferred by forward propagation are shown in the left figure, while the types inferred by the backward pass are shown in the right figure. We use* `32*u` *to denote a string of 32* ⟨`u`⟩ *bits. (We assume that a C* `short` *has 16 bits, while a* `char` *has 8.)*

Straight Line Code. We first analyze the program in Figure 2.[3] The algorithm begins with the forward pass and examines the first statement. First all the variables get a type with width specified by the C width, and all bits *don't know*, i.e. every bit is significant. `c+a` from Figure 2 must be computed on 9 bits, but result will be truncated to 8 bits of precision by taking ∪ with the *best* value. The masking operation creates a type for `d` with a combination of constants and *don't knows*, ⟨`00uu00uu`⟩.

The left shift in the return statement concatenates ⟨`0`⟩ bits at the least significant end, while the right shift's result will have the type ⟨`00uu`⟩. Using this information, the addition in the `return` statement infers that the final result has type ⟨`uu00uuuu`⟩.

The backward pass uses this information as a starting point. It proceeds to determine which bits of the computation are actually needed. In this example, the right shift indicates that the bottom 4 bits of `d` are *don't cares*, and the left shift indicates that the top 2 bits are *don't cares*. Since `d` is used in two expressions, its useful bits are represented by the ∩ of these two strings. The middle two bits of `d` have been found to be 0 by forward propagation, and they are preserved by taking the ∪ with the *best* value.

From the `&` we deduce that the useful bits of the sum `a+c` are ⟨`xxuuxxuu`⟩. This *don't care* information propagates up through the transfer function associated with the *plus* operation, and the compiler deduces that for both `a` and `c` only the bottom 6 bits are significant.

During the next forward pass there are no changes and the algorithm terminates.

Cycles. Figure 3 illustrates how loops are handled, requiring the algorithm to make several iterations. The forward pass discovers in the first iteration that the initial types of both `i` and `r` are ⟨`0`⟩. After the first addition to `r`, it still has the type ⟨`0`⟩. After processing the incrementation of `i`, however, it is noted that `i` must have type ⟨`1`⟩. The forward algorithm takes the ∩ of the two values discovered for `i` so far, ⟨`0`⟩ and ⟨`1`⟩, yielding

[3]We assume that all computations are carried on 8 bits; a normal C implementation would cast all values to `int` and back.

$\langle \texttt{u} \rangle$. When the assignment `r += i` is processed, `r` also gets the type $\langle \texttt{u} \rangle$.

The second iteration will assign to `i` the type of `i`+1, that is $\langle \texttt{u} \rangle + \langle 1 \rangle$, which is $\langle \texttt{uu} \rangle$. `r` will be assigned `r + i` which is $\langle \texttt{uu} \rangle + \langle \texttt{u} \rangle$, resulting in $\langle \texttt{uuu} \rangle$. Each additional iteration adds additional $\langle \texttt{u} \rangle$'s, for up to 16 iterations, at which point both `i` and `r` are labeled with 16 $\langle \texttt{u} \rangle$ bits and the algorithm terminates. (The length never becomes more than 16, because the values can never become "worse" in the lattice than the initial *best* value, which is given by the C type.)

The backward pass finds that the top 8 bits of `r` are *don't cares* because only a `char` is returned. This information propagates up through the `r += i` instruction, finally resulting in only 8 meaningful bits for `r`. This information also indicates that only 8 bits of `i` are useful in the computation of `r`. However, we cannot restrict the number of bits of `i` to 8 because `i` is also used in the comparison operation in the test of the `for` loop. The instruction `i < a` requires all of `i`'s bits to produce its result (i.e. it doesn't propagate any *don't cares* upwards). The backward pass takes the $\cap$ of all the types of the instructions using `i`: the comparison (16 $\langle \texttt{u} \rangle$'s) and the incrementation (8 $\langle \texttt{u} \rangle$'s), discovering that `i`'s type is 16 $\langle \texttt{u} \rangle$'s.

4 Experiments With a C Compiler

We evaluate our algorithm implemented in SUIF [16] on C programs from MediaBench [8] and SpecINT95 [13]. We run BitValue after all the important compiler optimizations. We use the basic SUIF compiler optimizations, augumented with a few optimizations of our own; these optimizations are sometimes less powerful than the ones available in commercial compilers, which can make BitValue look better.

We call a bit "useless" if it has a constant or *don't care* value. This bit brings no useful information at run-time. In the following we will present dynamic counts of the useless bits. The dynamic counts are obtained by using run-time profile information collected on a single input. The profile information is obtained by counting the execution frequency of each instruction, using instrumented binaries. Arguably, the dynamic count is more important than the static count, because it reflects the resources wasted by the computation. The dynamic count can potentially be translated into application speed-up.

BitValue is implemented as an iterative dataflow algorithm based on work-lists; it uses def-use chains [15]. We run BitValue intraprocedurally; an interprocedural implementation would improve the results, at the cost of greater compilation time. We do not analyze any of the library routines, and we do not include these in the dynamic counts. We treat library routines conservatively: their arguments and return values are all $\langle \texttt{u} \rangle$s.

4.1 Sensitivity to the Def-Use Analysis

Computing precise def-use information can be prohibitively expensive in the presence of arrays and pointers. To determine the sensitivity of our analysis to the precision of the def-use information, we compare the results of a simple intraprocedural def-use analysis with a sophisticated interprocedural analysis based on SPAN [12]. We implemented a def-use pass which assumes that all pointer operations, global variables and arrays alias to each

	Static	Dynamic
local	12	17
SPAN	15	20

Table 3: *Percent reduction in the number of useful bits computed; the numbers are the geometric mean for a few of the smaller benchmarks in the Mediabench benchmark suite.*

other; our analysis has a polynomial worst-case running time. SPAN is very precise, being based on whole program alias analysis, and has an exponential worst-case running time.

Table 3 shows the geometric mean of the saved bytes (in percents) for some of the benchmarks[4] for each of the two def-use analyses. More precise def-use information would enhance the quality of our algorithm by an additional 15%.

All the measurements we present in the subsequent sections use the fast and imprecise def-use analysis, with BitValue run on each procedure separately.

4.2 Range Analysis

The BitValue algorithm does not do a very good job on loop carried dependences. For a loop like `for (int i=0; i<2; i++)` the BitValue algorithm will infer a type of 32 ⟨`u`⟩s for `i`. However, from the loop bounds we can tell that two bits are enough to store its value.

To circumvent this problem we have also implemented a simplified variant of the bitwidth analysis algorithm described in [14]. This algorithm maintains for each integer quantity a range of possible values. The loop bounds are used to derive the bounds for loop induction variables. Dataflow analysis is used to derive the bounds for the other values.

When both analyses are run, BitValue and the range analysis can reinforce each other, discovering different sets of useless bits. The range analysis can only discover bits at the most significant side of a word, by design. When loop bounds are unknown, BitValue can be used to find approximate bounds for them, seeding the range analysis for the induction variables. Alternatively, as shown in the case above, the savings found using the range analysis for induction variables can be propagated by BitValue in the rest of the program.

In the following sections we present results which use both range and BitValue analysis. We ran three experiments for each benchmark: the range analysis only, BitValue only, and both. When we ran both analyses, they were alternated until a fixed point was reached, as shown by the pseudocode in Appendix B.

4.3 Evaluation

We are evaluating our algorithms on programs from the Spec95 integer benchmark suite and the Mediabench suite. In Mediabench some programs come in pairs encoder-decoder; we indicate them using a _e or _d suffix. The graph in Figure 4 displays the percent of the dynamic counts of useless bits, obtained using range analysis and BitValue analysis together. For each benchmark we have four different bars.

[4]The exponential running time precluded us from using the precise analysis on the larger programs.

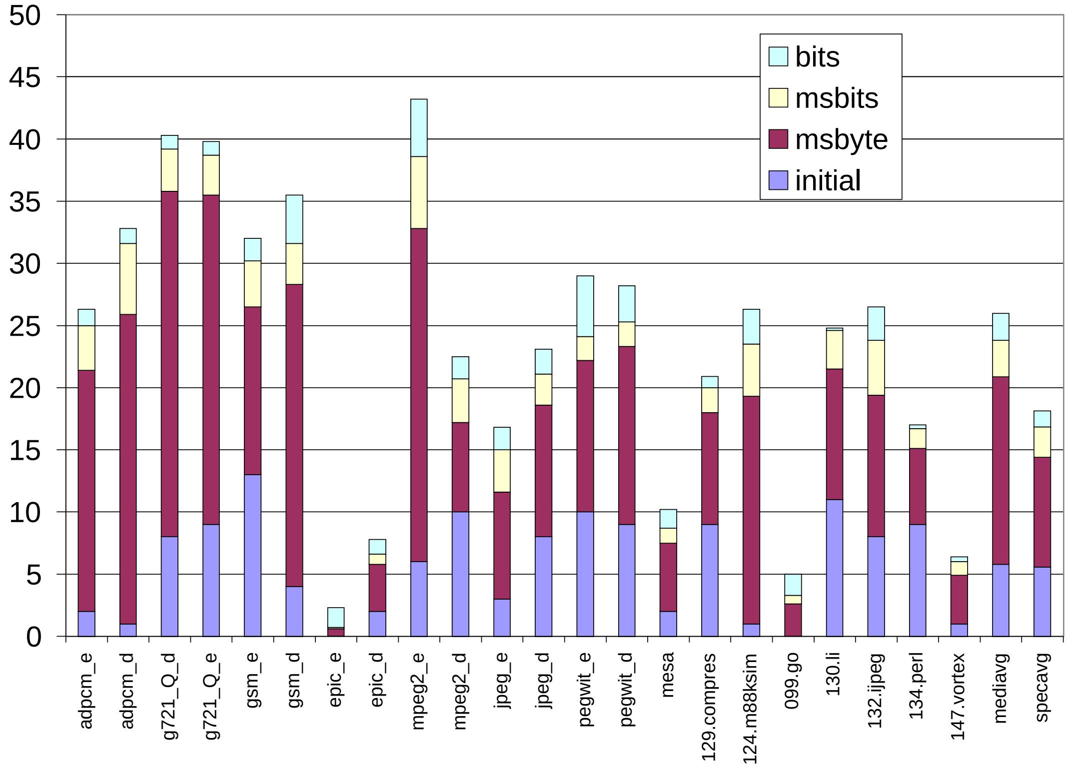

Figure 4: *Percent of the dynamically manipulated bits which are useless in programs from Mediabench and SPECInt95.*

- The bottom bar indicates bits which are saved by the C type declaration. For instance, if a value is declared to be a `short`, we say that it saves 16 bits (we assume a target machine with 32 bit word size).

- The second bar from the bottom counts only the *whole bytes* which are saved. Moreover, these bytes have to be present at the most significant part of the word. For a value having a ⟨`u`⟩ bit in the most significant byte of a word and all the other bits constant, we count no savings.

- The third bar additionally counts all the contiguous bits which are saved at the top of a word. For a type like ⟨`01x0u0x0`⟩ we count 4 saved bits.

- The topmost bar additionally counts all the other saved bits, no matter where they appear in the word.

The rightmost two bars are the arithmetic average for all the benchmarks in Mediabench and SPECInt95 respectively. There are almost no savings for the epic benchmarks because they operate with floating-point values in their innermost loops, being impermeable to our analysis.

In general, the more narrow values are present in the original program (i.e. the program is written using types shorter than `int`), the better our analyses perform, because they can propagate such information to the sources and destinations of the instructions using the narrow data.

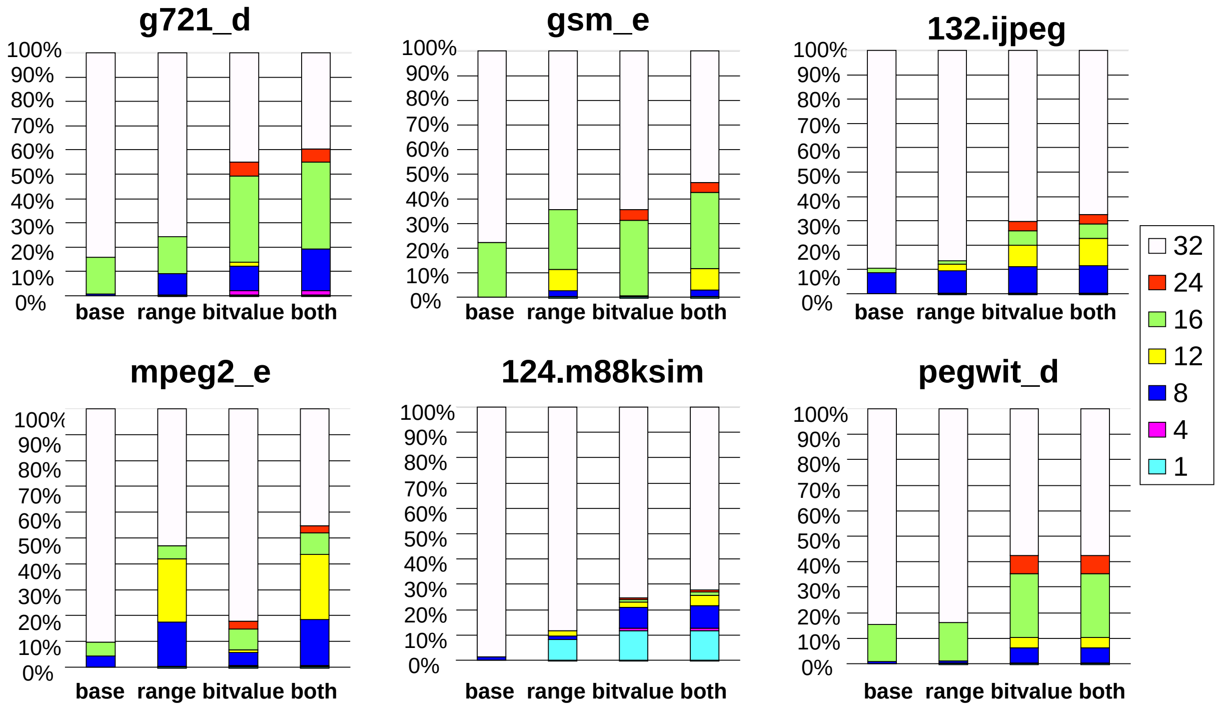

Figure 5: *Percentage breakdown of widths from some programs (dynamic counts). The 'base' bar is the original program; 'range' indicates the distribution after range analysis, 'bitvalue' after running BitValue and 'both' after both analyses were run.*

These results are roughly the same as the ones presented in [5]. However, there are three differences with respect to that paper: (1) we have implemented a better dead-code ellimination procedure, which is run before our analysis. This reduces the visible benefit of our analyses, (2) we have improved the range analysis to deal with more instructions, and (3) we alternate the range analysis with BitValue. This provides better results, because the two analyses reinforce each other. The loss in savings due to the better dead-code ellimination is ballanced by the more savings we find.

4.4 Size Histograms

Table 4 and Table 5 show the histograms of the data sizes for our benchmarks. The value sizes are bucketed in bins as follows: 1 bit, 2–4 bits, 5–8 bits, 9–16 bits, 17–24 bits and 25–32 bits wide values. We count only useless bits from the most significant part. For each program we present four histograms: one for the original program (with no analysis), one for the range analysis alone, one for BitValue analysis alone, and one for both analyses.

In Figure 5 we show the same information graphically for some benchmarks which achieve most savings. For example, we can interpret the graphs for `g721_d` in the following way: the first bar says that about 16% of the values in the original program are 16-bit or less. The fourth bar shows that using the combined analyses we discover that 16 bits are actually enough for about 55% of the values in the program.

For some programs the range analysis finds most useless bits, for some programs BitValue is more effective, but in general combining the two gives results better than any of them isolated.

Benchmark	Experiment	1	4	8	12	16	24	32
adpcm_e	Original	0	0	0	0	4	0	96
	Range	4	0	4	0	4	0	88
	BitValue	4	13.4	8	0	4	0	70.6
	Both	4	13.4	8	0	4	0	70.6
adpcm_d	Original	0	0	0.9	0	1.9	0	97
	Range	3.9	0	2.9	0	2	0	91.2
	BitValue	5.9	17.3	6.9	0	2	0	67.9
	Both	5.9	17.3	6.9	0	2	0	67.9
g721_d	Original	0	0	0.6	0	15.3	0	84.1
	Range	0.2	0	8.8	0	15.3	0	75.7
	BitValue	0.2	1.9	10.1	1.7	35.5	5.7	44.9
	Both	0.2	1.9	17	0.3	35.6	5.5	39.4
g721_e	Original	0	0	0.4	0	15.8	0	83.8
	Range	0.1	0	7.8	0	15.7	0	76.1
	BitValue	0.2	1.9	9.1	1.6	36.7	5.8	44.7
	Both	0.2	1.9	15.3	0.3	36.8	5.7	39.8
gsm_e	Original	0	0	0	0	22.2	0	77.8
	Range	0.3	0	2.4	8.8	24.2	0	64.3
	BitValue	0.3	0.1	0.2	0	30.7	4.3	64.4
	Both	0.3	0.1	2.5	8.8	30.8	4.2	53.2
gsm_d	Original	0	0	0.1	0	19.3	0	80.6
	Range	1.8	0	2	6.9	21.8	0	67.1
	BitValue	1.9	0.2	0.2	0	30.6	5.9	61.2
	Both	1.9	0.1	2.2	7.1	30.7	5.8	52.1
epic_e	Original	0	0	0.7	0	0.2	0	99.1
	Range	0	0	0.6	0	0.1	0.2	98.7
	BitValue	0	0	0.9	0	0.7	0.1	97.9
	Both	0.1	0	1	0	0.8	0.4	97.7
epic_d	Original	0	0	4.4	0	2.1	0	93.5
	Range	2.1	0	4.4	0	2.1	1	90.4
	BitValue	2.1	0	6	0	5.2	0.6	86.1
	Both	2.1	0	6	0	5.2	1.6	85.1
mpeg2_e	Original	0	0	4.5	0	5.2	0	90.3
	Range	0.5	0	17.1	24.3	5.2	0	52.9
	BitValue	0.5	0.1	5.1	1.1	7.9	3.1	82.2
	Both	0.5	0.1	17.7	25.4	8.2	2.7	45.4
mpeg2_d	Original	0	0	7	0	7.6	0	85.4
	Range	0.6	0	7.4	1.8	7.6	0	82.6
	BitValue	0.8	0.1	7.5	2	10.2	6.7	72.7
	Both	0.8	0.1	7.9	3.8	11.6	5.2	70.6
jpeg_e	Original	0	0	2.5	0	3.1	0	94.4
	Range	2	0	2.6	2.9	3.1	0	89.4
	BitValue	2.1	0.2	3.8	4.4	4.8	3.9	80.7
	Both	2.1	0.2	3.9	7.4	4.8	3.9	77.7

Table 4: *Percentage breakdown of widths (to continue).*

The percentage of values which are less than 16 bits averaged over all the programs when both analyses are applied is 26.8%. Simulation studies [3] have shown that for this benchmark mix (and for a fixed input) around 50% of the values computed are less than 16 bits; our static analysis is able to discover almost half of these. Because our analysis

Benchmark	Experiment	1	4	8	12	16	24	32
jpeg_d	Original	0	0	4.3	0	2.7	0	92.9
	Range	1.5	0	4.4	2.1	2.7	0	89.2
	BitValue	1.4	0.1	5.8	5.6	5.4	3.5	77.6
	Both	1.5	0.2	6	7.8	5.5	3.6	75.4
pegwit_e	Original	0	0	0.3	0	16.1	0	83.4
	Range	0.4	0	0.4	0	16.2	0	83
	BitValue	0.7	0	3.7	2.4	26.3	6.6	59.8
	Both	0.8	0	3.8	2.5	26.4	6.6	59.9
pegwit_d	Original	0	0	0.6	0	14.9	0	84.5
	Range	0.3	0	0.6	0	14.8	0	84
	BitValue	0.4	0	5.6	4.3	25.1	6.9	57.7
	Both	0.4	0	5.6	4.3	25.1	6.9	57.7
mesa	Original	0	0	2.8	0	0.8	0	96.4
	Range	0.9	0	2.8	0	0.9	2.4	93
	BitValue	2.8	0.1	3.7	0.2	1.4	4	87.8
	Both	2.8	0.1	3.7	0.2	1.5	6.4	85.3
129.compress	Original	0	0	8.1	0	1.7	0	90.2
	Range	3.2	0	8.1	0.5	1.7	0.9	85.6
	BitValue	3.2	3.2	11.3	0.6	1.7	0.7	79.2
	Both	3.2	3.2	11.3	1.2	1.9	1.6	77.5
124.m88ksim	Original	0	0	1.3	0	0	0	98.7
	Range	8.1	0	1.6	1.8	0	0	88.4
	BitValue	11.5	1.1	8.3	1.9	1.2	0.6	75.3
	Both	11.5	1.2	9	4.1	1.2	0.6	72.4
099.go	Original	0	0	0	0	0	0	100
	Range	3.3	0	0	0	0	0	96.7
	BitValue	3.3	0	0.1	0	0	0	96.6
	Both	3.3	0	0.1	0	0	0	96.5
130.li	Original	0	0	8.1	0	0	0	91.9
	Range	12.7	0	8.1	0	0	0	79.2
	BitValue	13.2	0	8.1	0	0	0	78.7
	Both	13.2	0	8.1	0	0	0	78.7
132.ijpeg	Original	0	0	8.9	0	1.7	0	89.4
	Range	0.4	0	9.1	2.6	1.7	0	86.2
	BitValue	0.3	0	10.7	8.6	5.8	3.7	70.3
	Both	0.3	0	10.9	11.2	5.8	3.7	67.5
134.perl	Original	0	0	7.5	0	2.3	0	90.2
	Range	5.5	0	7.5	0	2.3	0	84.7
	BitValue	6.3	0.1	8.2	0.3	2.6	0	82.4
	Both	6.3	0.1	8.2	0.3	2.6	0	82.4
147.vortex	Original	0	0	0.5	0	2.6	0	96.9
	Range	3.9	0	0.5	0	2.6	0	93
	BitValue	3.9	0.4	0.9	0	2.6	0.9	91.3
	Both	3.9	0.4	0.9	0	2.6	0.9	91.3

Table 5: *Percentage breakdown of widths (continued).*

does not deal with arrays and accesses through pointers and because our results are valid for any input data, we consider that these results are very strong.

In fact, for some benchmarks we discover a larger number of values less than 16 bits than [3]y. There are two reasons for this: (1) the Suif compiler is less aggressive in optimizations than the Alpha compiler used for that study, so we may execute extra instructions on narrow values, which bias the dynamic count in our favor; (2) that study reports percents of instructions whose both inputs are less than 16 bits, but we report instructions whose output is less than 16 bits. These quantities are not neccessarily the same, but should be close.

We have examined the main sources of reductions to gain insight into the effectiveness of the algorithm. The sources of reduction found by BitValue come from several patterns: (1) the use of shift, bitwise "and" and "or" and multiplication by small constants; (2) the propagation of cast information involving narrow types both forward and backward; (3) array index computations and (4) loop induction variables for FORTRAN-like DO-loops.

The data show that BitValue may be a very useful ingredient for automatic compilation for MMX-like parallelism. It is not clear though how many of the narrow values we discover can be exploited by the scheduler of the compiler by being packed together.

These qualitative results encourage us to continue the exploration of computer architectures and compiler algorithms which can effectively exploit narrow data values.

4.5 Practical Issues

Our implementation of BitValue is fast and scales in practice linearly with program size. The space complexity is linear, too. We analyze on average 1000 lines/second on a 750Mhz PIII (excluding file I/O and def-use computations), with an untuned implementation.

BitValue effectively generalizes constant folding and dead-code elimination: the constant values discovered by classical constant-folding algorithms will be a subset of the constant values discovered by BitValue, and the dead code will be a subset of the instructions discovered by BitValue as having the output type $\langle \mathtt{x} \rangle$. BitValue can potentially discover more instances of constants and dead-code than usual algorithms. Actually BitValue's dead code discovery algorithm is more powerful than a simple-minded approach based on pruning the instructions which have no users in the def-use chain: BitValue effectively discovered cycles of instructions whose result is not visible globally, but which use each other's results.

We have validated our implementation by self-instrumenting the programs. After detecting the type for each value, we have inserted operations to mask away the constant $\langle \mathtt{0} \rangle$ bits, to set the constant $\langle \mathtt{1} \rangle$ bits and we gave a random value to the $\langle \mathtt{x} \rangle$ bits. The modified programs were run to check the validity of the output.

An interesting side-effect of our analysis is that it gives a portable high-level method for specifing widths: by using a masking operation the programmer can seed the BitValue algorithm. For example, the statement `c = c & 0x3c` indicates that only the middle 4 four bits of `c` are useful, and this knowledge is propagated by BitValue throughout the code.

We are now incorporating BitValue in a compiler which automatically extracts configurations from C programs for execution in a mixed environment, consisting of a CPU augumented with a reconfigurable fabric. Preliminary results indicate that BitValue will enable non-trivial speed-ups.

5 Experiments with a Reconfigurable Hardware Compiler

In this section we evaluate the BitValue algorithm as it is used in the DIL compiler [4] developed for compiling to reconfigurable hardware. The DIL language operates on arbitrary-precision integer data types and does not require the values to be annotated with an explicit width.

With one exception, the algorithm used in the DIL compiler is essentially the same as that used in the C compiler. Because it allows unbounded precision, the DIL compiler tries to statically bound the precision of the manipulated values, to ensure the finiteness of the lattice. There are cases when this is not possible (for instance when the program cannot be approximated by any finite-precision program), and then the compiler asks the user to give explicit bounds for the variables.

5.1 Reconfigurable Hardware Benchmarks

In order to evaluate BitValue with DIL, We analyze a set of kernels typical for reconfigurable hardware systems, shown in Table 6, and described in more detail in [7].

Benchmark	**Description**
Cordic	12 stage implementation of Cordic vector rotations.
DCT	One-dimensional, 8-point discrete cosine transform.
Encoder	8-bit Huffman encoder with the code table hardwired.
FIR	FIR filter with 20 taps and 8-bit coefficients.
IDEA	Complete 8 round International Data Encryption Algorithm with the key compiled into the configuration.
Nqueens	Evaluator for the n-queens problem on an 8x8 board.
Over	Porter-Duff “over” operator.
Popcount	Count the number of “1” bits in a 16-bit word.

Table 6: *Benchmark kernels used to evaluate the DIL compiler.*

5.2 Size Reductions for DIL Programs

Because of the nature of DIL, there is no baseline for comparing the performance of the algorithm (in C we could compare the reduced sizes with the C type-specified sizes). For evaluation purposes we artificially set the sizes of all variables to 32-bits[5] and then we run the algorithm to determine the reduction in size.

We examine two architectures: one which uses 8-bits wide processing elements (PEs) and another which maps the program to a circuit with 1-bit PEs. The former is prototypical of more recent reconfigurable devices, the latter of commercially available field programmable gate arrays.

[5] Our C implementations of these kernels, used for evaluating their performance on a UltraSparc, were written in this way [7]. Even if this methodology may be considered “unfair” because it starts from a very large baseline circuit, it still evaluates the capacity of our algorithms to detect useless bits.

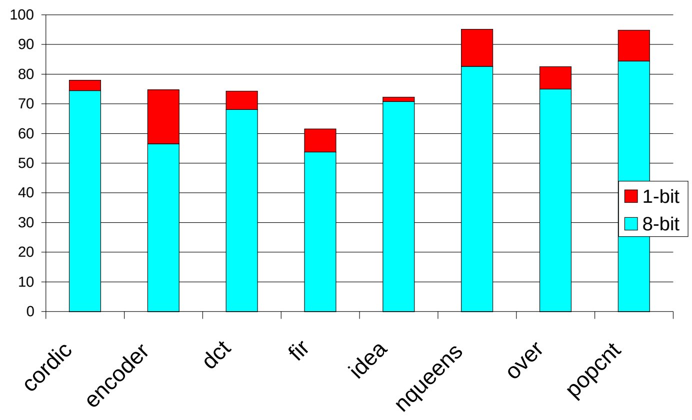

Figure 6: *Percent of the hardware removed by BitValue when synthesizing programs for reconfigurable hardware devices; the bottom bar shows savings when using 8-bit wide processing elements, and the top bar when using 1-bit wide processing elements.*

Figure 6 shows the reduction in the amount of hardware required to implement kernels compiled with the DIL compiler. For 8-bit PEs, the compiler cannot optimize away all useless bits: an $\langle \mathtt{x} \rangle$ in the middle of a byte of $\langle \mathtt{u} \rangle$ bits cannot be removed (but we can remove all bytes with only useless bits, irrespective of their position inside of the word). On the other hand, for 1-bit targets, the hardware reductions are maximum, because none of the bits discovered by the compiler as useless must be implemented.

Note that the impact of the analysis is significant: it can decrease the silicon real-estate (and implicitly, decrease the power consumption and the latency of the computation) by a factor between 2 and 20. For PipeRench [7], a reconfigurable device developed at CMU, any reduction in size translates immediately to higher speed.

5.3 Practical Issues

Compiling for reconfigurable devices is usually done with CAD tools, which use Boolean function manipulation techniques to simplify the circuits. Although our reductions are intrinsically less powerful than the algorithms based on Boolean functions, they generate results of acceptable quality and run several orders of magnitude faster. The techniques using Boolean simplifications often have worst-case exponential complexity, while our algorithm has a theoretical quadratic run-time.

To compare our algorithm for DIL with the ones of commercial CAD tools, we ran the DCT kernel through BitValue and the Synopsis Sinplify compiler; we report these results also in [4]. We find that our analysis pass runs two orders of magnitude faster (a few seconds compared to tens of minutes). When targeted to 1-bit processing elements, our analysis produces a circuit with 2654 bit operations. The Synopsis tools produce a circuit with 899 Xilinx 4xxx CLBs. A CLB is equivalent to 2–3 bit-operations, depending on how it is being used. Thus our analysis yields results close to the more complicated analysis of Synopsis (within 30%). Note that the Synopsis result is sensitive to the programmer's width specifications whereas our BitValue algorithm infers the widths automatically (the circuit given to Synopsis was using the best estimates the programmers could manually produce for the width of the values).

6 Related Work

There is a wealth of static and dynamic analyses which suggest that many of the bits computed by a program are useless.

Brooks and Martonosi [3] use a simulator to show that for the programs in both SpecInt95 and MediaBench more than half of all integer computations require at most 16 bits of precision. Our compile time analysis proves statically that on average 27% of the widths are 16 bits or less for any input data. [3] also suggests hardware techniques for creating instructions which operate on narrow widths on the fly. The work of Bondalapati and Prasanna is similar, looking at dynamically changing functional unit sizes based on dynamically maintained width information [2].

Static techniques for inferring minimum bit-widths using *don't care* detection are prevalent in the logic synthesis community (for example [6]). This approach computes *satisfiability don't care* sets on a network of Boolean operators. Such an analysis operates at the bit (and not at the word) level and is significantly slower but more precise than our approach. These algorithms are exponential in complexity, and even heuristic methods cannot address benchmarks of the size we are analyzing, while our algorithm has worst-case quadratic complexity, and linear complexity in practice. We compared our algorithm to the Synopsis Synplify compiler, a commercial CAD tool, using the DCT benchmark from Section 5. Our analysis runs two orders of magnitude faster and generates circuits within 30% of the size obtained by Synopsis.

Most similar to our work is Razdan [11]. His analysis uses a ternary logic of 0, 1 and *don't know* (denoted in this paper by `x`); he also operates on strings of bits, and uses forward and backward analyses. Although he handles loop induction variables for loops with a statically known trip-count, he does not offer a complete solution for handling loop-carried dependences, where a lot of savings can be gained.

Babb et al. [1] suggest that width analysis can be performed by determining the maximum values that can be carried on the wires, for example by examining loop bounds. This technique is further investigated by Stephenson et al. in [14]. These techniques are orthogonal to ours. The technique in [14] was re-implemented in a simplified form by us as the range analysis[6]; such a technique works better for loop carried dependences when the loop bounds are know. By combining this analysis with BitValue we can obtain savings even for loops where the bounds are not known at compile time.

7 Conclusions

We have presented BitValue, a compiler algorithm which infers statically the values of the bits computed by a program. Trimming constant bits or unused bits can reduce the width of the computed values, enabling the compiler to use narrow width functional units, which have become available in new architectures (e.g. MMX, reconfigurable functional units, and Application-Specific Instruction Processors).

[6] [14] does a more sophisticated loop induction variable detection. It also implements a backward propagation, and relies on full-program alias analysis to analyze pointer and array data at the expense of increased compilation cost.

BitValue can be used to analyze both C and DIL programs to significantly reduce the number of bits used to perform computations. We show that BitValue inference can determine that on average 14% of the most significant bytes (and 20% of the bits) computed are unnecessary for programs from MediaBench and SpecINT95. BitValue analysis can reduce the size of the programs synthesized for a reconfigurable architecture between two- and twenty-fold. The algorithm we present is an essential ingredient in developing a compiler which will target sub-word parallel media extensions, low power extensions, or reconfigurable devices.

A The BitValue Dataflow Algorithm

Here is the pseudocode implementation of the BitValue dataflow algorithm. For brevity we assume that each instruction has only one input and one output. The implementation of the forward_transfer and backward_transfer functions is shown in Appendix C.

procedure initialize
begin
    foreach value v
        best(v) = v's C type as bitstring
end

procedure clear
begin
    foreach value v
        if (not is_input(v)) current(v) = $\top$
        else current(v) = best(v)
end

procedure mix
begin
    foreach value v
        best(v) = best(v) $\cup$ current(v)
end

procedure forward
begin
    while (some current changed)
        foreach instruction i
            $u = \top$
            foreach definer d of input(i)
                $u = u \cap$ current(d)
            current(output(i)) = forward_transfer(i, u) $\cup$ best(output(i))
    end while
end

```
procedure backward
begin
    while (some current changed)
        foreach instruction i
            u = ⊤
            foreach user d of output(i)
                u = u ∩ current(d)
            current(input(i)) = backward_transfer(i, u) ∪ best(input(i))
    end while
end

procedure bitvalue
begin
    initialize()
    while (best changed in last operation)
        clear()
        forward()
        mix()
        clear()
        backward()
        mix()
    end while
end
```

B The Width Analysis Algorithm

The width analysis is carried by alternating BitValue with a range analysis [14]. The range of a variable is represented as an interval of two integer values, which are the minimum and maximum values that the variable can reach during any execution of the program. We use two auxilliary procedures which can convert ranges to bitstrings and viceversa.

```
procedure convert_intervals_to_types
begin
    foreach value v
        u = interval_to_type(best_interval(v))
        best(v) = best(v) ∪ u
end

procedure convert_types_to_intervals
begin
    foreach value v
        u = type_to_interval(best(v))
        best_interval(v) = best_interval(v) ∪ u
end
```

```
procedure width
begin
    repeat
        interval_analysis()
        change = convert_intervals_to_types()
        bitvalue()
        change = change or convert_values_to_intervals()
    while (some change in best or best_interval)
end
```

C Implementation of the transfer functions

In this section we give pseudo-code for our implementations of the transfer functions. As described in Section 2, we only implement conservative approximations to the "best" transfer functions[7].

We use a few auxilliary procedures and constants, which are not shown in detail. The leading zero bit of the constants below is useful to represent signed magnitudes:

signExtend: implements sign extension of a bitstring to a specified length, by either padding it with ⟨`0`⟩ for unsigned values or by duplicating the most significant digit for signed values.

equalizeLength: brings two bitstrings to the same length by sign-extending the shorter one.

allunknown(*length*): returns a bitstring with all bits ⟨`u`⟩ of the given length.

True: is ⟨`01`⟩

False: is ⟨`0`⟩

Dontknow: is ⟨`0u`⟩

Many operations can be described by a table; in these cases the operation is implemented bit by bit. **equalizeLength** is invoked first, to bring the two bitstrings to the same length. All type computations first check if the arguments represent constant values; if so, they use the native arithmetic of the machine to carry the computation and convert the resulting value back to a bitstring.

C.1 Forward Transfer Functions

- `inf(a, b)`:

[7] A best reverse transfer function may not even exist for the reverse data-flow analysis.

a	b			
	0	1	u	x
0	0	u	u	0
1	u	1	u	1
u	u	u	u	u
x	0	1	u	x

- `sup(a,b)`: bring longest bitstring to length of shorter

a	b			
	0	1	u	x
0	0	x	0	x
1	x	1	1	x
u	0	1	u	x
x	x	x	x	x

- `a+b+carry` [carry can never be ⟨`x`⟩]

a	b	carry		
		0	1	u
x	x	0x	01	0u
x	0	00	01	0u
x	1	01	10	uu
x	u	0u	uu	uu
0	x	00	01	0u
0	0	00	01	0u
0	1	01	10	uu
0	u	0u	uu	uu
1	x	01	10	uu
1	0	01	10	uu
1	1	10	11	1u
1	u	uu	1u	uu
u	x	0u	uu	uu
u	0	0u	uu	uu
u	1	uu	1u	uu
u	u	uu	uu	uu

- `a-b-borrow` [borrow can never be ⟨`x`⟩]

a	b	borrow		
		0	1	u
x	x	00	00	00
x	0	00	00	00
x	1	00	10	uu
x	u	00	0u	uu
0	x	00	01	uu
0	0	00	11	uu
0	1	11	10	1u
0	u	uu	1u	uu
1	x	00	00	0u
1	0	01	00	0u
1	1	00	11	uu
1	u	0u	uu	uu
u	x	0u	uu	uu
u	0	0u	uu	uu
u	1	uu	1u	uu
u	u	uu	uu	uu

- `a * b`:
  - check if one operand is a constant power of 2 and concatenate ⟨`0`⟩s at the end of the other one
  - ta = no of trailing zeros of `a`; tb = no of trailing zeros of `b`
  - la = no of leading zeros of `a`; lb = no of leading zeros of `b`
  - return **allunknown**(length(`a`) + length(`b`) $-ta-tb-la-lb$) concatenated with $(ta+tb)$ ⟨`0`⟩s.
- `a | b`:

a	b			
	0	1	u	x
0	0	1	u	x
1	1	1	1	1
u	u	1	u	x
x	x	1	x	x

- `a ^ b`:

a	b			
	0	1	u	x
0	0	1	u	x
1	1	0	u	x
u	u	u	u	x
x	x	x	x	x

- `a & b`:

a	b			
	0	1	u	x
0	0	0	0	0
1	0	1	u	x
u	0	u	u	x
x	0	x	x	x

- `a && b`:
  - if `a` and `b` have ⟨`1`⟩ bits return **False**
  - if `a` or `b` has ⟨`u`⟩ bits return **Dontknow**
  - return **False**
- `a || b`:
  - if either `a` or `b` have ⟨`1`⟩ bits return **True**
  - if `a` or `b` has ⟨`u`⟩ bits return **Dontknow**
  - return **False**
- `a == b` = `! (a != b)`
- `a != b`
  - if some bit of `a` or `b`is ⟨`u`⟩ return **Dontknow**
  - if all bits are constant and two corresponding bits are different return **True**
  - return **False**
- `a < b`: bring `a` and `b` to same length by **signExtension**; scan bits starting from most significant and for each bit test:
  - if `a`'s or `b`'s bit is ⟨`u`⟩ return **Dontknow**;
  - if `a`'s bit is ⟨`0`⟩ and `b`'s is ⟨`1`⟩ return **True**;
  - if `b`'s bit is ⟨`0`⟩ and `a`'s is ⟨`1`⟩ return **False**;

  return **False**;
- `a > b` = `b < a`
- `a <= b` = `!(a > b)`
- `a >= b` = `!(a < b)`
- `a % b` return **allunknown**(min(length(`a`), length(`b`)))
- `a / b` return **allunknown**(length(`a`))
- `a << b`

- if (`b` is constant) return `a` concatenated with `b` ⟨`0`⟩s
- else return **allunknown**(length(`a`) concatenated with $2^{\text{length}(\texttt{b})}$)

- `a >> b`
  - if (`b` is constant) return `a` without its bottom `b` bits
  - else return **allunkown**(lenght(`a`))

- `!a`
  - if `a` has a ⟨`1`⟩ bit return **False**
  - if `a` has a ⟨`u`⟩ bit return **Dontknow**
  - return **True**

- `~a`

a			
x	0	1	u
x	1	0	u

- `(signed)a`:
  return ⟨`0`⟩ concatenated with `a`

- `(unsigned)a`:
  [is supplied *width* of the result as argument] return **signExtend(**`a`**,** *width***)**

- `a` cast to a different width:
  - if width is enlarged by cast return `a`
  - else truncate `a` to output width

C.2 Backward Transfer Functions

If a function is not indicated, or if some case is not treated, that operation propagates no don't cares from the output to the input. The only exception is when *all* bits of the output are don't cares; then they all propagate to all inputs (the output is dead code).

- all carry operations (`+, -, *`):
  truncate all ⟨`x`⟩ bits from the most significant end

- `a | b = c`: for each bit of `c`
  - if the output is ⟨`x`⟩, this input is also ⟨`x`⟩
  - if the other input is ⟨`1`⟩ and this input is not ⟨`1`⟩, this input is ⟨`x`⟩
  - else this input is ⟨`u`⟩

- `a & b = c`: for each bit of `c`
  - if the output is $\langle$`x`$\rangle$, this input is also $\langle$`x`$\rangle$
  - if the other input is $\langle$`0`$\rangle$ and this input is not $\langle$`0`$\rangle$, this input is $\langle$`x`$\rangle$
  - else this input is $\langle$`u`$\rangle$
- `a >> b = c`

  if (`b` is constant) `a`'s don't cares = (`c` concatenated with `b` $\langle$`x`$\rangle$'s)
- `a << b = c`

  if (`b` is constant) `a`'s don't cares = `c` with `b` least significant bits removed

For these functions the output don't cares are exactly copied to the input:

- bitwise complementation
- bitwise xor
- casts

For these functions no don't cares are propagated to the input:

- all comparisons
- division, remainder

References

[1] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *IEEE/FCCM Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1999. MIT.

[2] K. Bondalapati and V.K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In *IEEE/FCCM Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1999. Organization: University of Southern California.

[3] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA-5*, January 1999. Princeton University.

[4] M. Budiu and S.C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *ACM/FPGA Symposium on Field Programmable Gate Arrays*, Monterey, CA, 1999.

[5] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of 6th International Euro-Par Conference*, Lecture Notes in Computer Science 1900, Springer Verlag, August 2000.

[6] M. Damiani and G. de Micheli. Don't care specifications in combinational and synchronous logic circuits. In *IEEE Transactions on CAD/ICAS*, pages 365–388, 1992.

[7] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

[8] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[9] P. Marwedel and G. Goossens, editors. *Code generation for embedded processors.* Kluwer Academic Press, 1995.

[10] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[11] Rahul Razdan. *PRISC: Programmable reduced instruction set computers*. PhD thesis, Harvard University, May 1994.

[12] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation*, 1999.

[13] http://www.specbench.org/osg/cpu95/.

[14] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.

[15] E. Stoltz, M. P. Gerlek, and M. Wolfe. Extended SSA with Factored Use-Def chains to support optimization and parallelism. In *Proceedings Hawaii International Conference on Systems Sciences*, Maui, Hawaii, Jan. 1994.

[16] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.