BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations*

Mihai Budiu[1], Majd Sakr[2], Kip Walker[1], and Seth C. Goldstein[1]

[1] Carnegie Mellon University
{mihaib,kwalker,seth}@cs.cmu.edu
[2] Pittsburgh University
sakr@ee.pitt.edu

Abstract. We present a compiler algorithm called BitValue, which can discover both 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 which target special processor architectures for computing on non-standard bitwidths.
Using this algorithm we show that up to 31% of the computed bytes are thrown away (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.

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 to be performed on operands which are smaller than the natural word size.

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

Using the special architectural features requires the programmer to use macro libraries or specify the bit-widths manually, a tedious and error-prone process. Furthermore, this is often impossible as there is little or no support in high-level languages for specifying arbitrary bit-widths.

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

In this paper we present the BitValue 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 inputs to the program (constant bits) and bits which do not influence the output of the program (unused bits). By eliminating computation 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 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.

In Section 2 we present our BitValue inference algorithm with an example. Results for the implementation in a C compiler are in Section 3 and for a reconfigurable hardware compiler in Section 4. Related work is presented in Section 5 and we conclude in Section 6.

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 the execution of the program. The BitValue algorithm solves this problem using dataflow analysis.

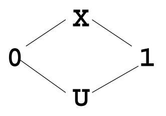

Fig. 1. *The bit values lattice.*

We represent the bit values by one of: $\langle \mathtt{0} \rangle$, $\langle \mathtt{1} \rangle$, *don't know* (denoted by $\langle \mathtt{u} \rangle$) and *don't care*, (denoted by $\langle \mathtt{x} \rangle$). 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, $\langle \mathtt{0} \rangle$ or $\langle \mathtt{1} \rangle$. A bit is labeled $\langle \mathtt{x} \rangle$ if it does not affect the output; otherwise a bit is labeled $\langle \mathtt{u} \rangle$. 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 is $\langle \mathtt{x} \rangle$ and the bottom is $\langle \mathtt{u} \rangle$.

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 `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 $\langle\texttt{uuuuxxxx}\rangle$.

The bitstrings also form a lattice $\mathcal{L}$. Space considerations preclude us from giving the formal definition of the operations on this lattice. The $\cup$ and $\cap$ operations in $\mathcal{L}$ are done bitwise (i.e. $ab \cup cd = (a \cup c)(b \cup d)$). 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.

The Transfer Functions. In order to certify the correctness of our algorithms, we need to prove that our transfer functions are monotone and conservative. For this purpose we provide mathematical definitions for the "best" forward transfer function and for a conservative backward transfer function.

We now define $\mathcal{A}$, the forward transfer function of an operator in $\mathcal{L}$. We define the auxiliary "expansion" function $\exp : \mathcal{L} \times \{\langle\texttt{x}\rangle, \langle\texttt{u}\rangle\} \to 2^{\mathcal{L}}$, which takes a bitstring s and a bit value $b \in \{\langle\texttt{x}\rangle, \langle\texttt{u}\rangle\}$, and generates a set of bistrings: all bitstrings that can be obtained from s by replacing the bits in s having the value b by all possible combinations of constant values. For example $\exp(\langle\texttt{0ux1x}\rangle, \langle\texttt{x}\rangle) = \{\langle\texttt{0u010}\rangle, \langle\texttt{0u110}\rangle, \langle\texttt{0u011}\rangle, \langle\texttt{0u111}\rangle\}$.

We now define three auxiliary functions which are used to compute the transfer function of any operator. $\mathcal{A}_c : (\mathbb{N} \to \mathbb{N}) \times \{0,1\}^* \to \mathcal{L}$ operates on "constant" bitstrings, i.e. bistrings containing only $\langle\texttt{0}\rangle$ and $\langle\texttt{1}\rangle$. $\mathcal{A}_u : (\mathbb{N} \to \mathbb{N}) \times \{\langle\texttt{0}\rangle, \langle\texttt{1}\rangle, \langle\texttt{u}\rangle\} \to \mathcal{L}$ computes the transfer function for bitstrings which comprise $\langle\texttt{0}\rangle$, $\langle\texttt{1}\rangle$ and $\langle\texttt{u}\rangle$ bits. $\mathcal{A} : (\mathbb{N} \to \mathbb{N}) \times \mathcal{L} \to \mathcal{L}$ works for bitstrings with any of the digits in $\mathcal{B}$. Given a unary operation $f : \mathbb{N} \to \mathbb{N}$, $\mathcal{A}(f, \cdot)$ is its associated forward transfer function in $\mathcal{L} \to \mathcal{L}$.

$$\begin{array}{lll}
\mathcal{A}_c(f,v) &= f(\text{value}(v)) & \text{where } v \in \{\langle\texttt{0}\rangle, \langle\texttt{1}\rangle\}^* \\
\mathcal{A}_u(f,v) &= \bigcap_{y \in \exp(v,\langle\texttt{u}\rangle)} \mathcal{A}_c(f,y) & \text{where } v \in \{\langle\texttt{0}\rangle, \langle\texttt{1}\rangle, \langle\texttt{u}\rangle\}^* \\
\mathcal{A}(f,v) &= \bigcup_{y \in \exp(v,\langle\texttt{x}\rangle)} \mathcal{A}_u(f,y) & \text{where } v \in \mathcal{L}.
\end{array}$$

The intuition behind these equations is the following: when we compute the transfer function in $\mathcal{L}$ for an input value, we can choose arbitrary values for the input bits which are marked $\langle\texttt{x}\rangle$, but we must search the entire space of possibilities for the bits marked $\langle\texttt{u}\rangle$. 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 ~ operator when applied to $\langle\texttt{u0x}\rangle$:

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

The backward transfer function will discover *don't care* bits in the input starting from the *don't care*s in the output. We do not have a closed form for the backward transfer function. We can, however, define a conservative approximation using techniques from Boolean function minimization [6]. The notion of *don't care* input for a Boolean function f of n variables is well known (x_i is a *don't care* if $\frac{\partial f}{\partial x_i} = 0$).

We can view an operator which computes many bits (like addition) as a vector of Boolean functions, each computing one bit of the result. An input bit is don't care for the operator if it is a don't care for *all* the functions in the vector whose result is not $\langle \texttt{x}\rangle$. If our analysis discovers that some input bits are constant, we can use those in the backward transfer function computation, starting the computation with the restriction of f to those constant inputs.

For example, let us see how the backward propagation operates on the statement `c = a^b` when we know already 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 of `c` to be propagated to `a` and `b`. The two bits of `c` are computed by two boolean functions of 4 bits: f_0 and f_1: $f_0(a_0 = 0, a_1, b_0, b_1) = a_0\hat{}b_0$ and $f_1(a_0, a_1, b_0, b_1) = a_1\hat{}b_1$. Because the bit 1 in the result is $\langle \texttt{x}\rangle$, we only need to look for the *don't care*s of f_0 which will be the *don't care*s of the input. For instance, bit a_1 is a *don't care*, because $f_0|_{a_1=0} = f_0|_{a_1=1}$, and so is b_1. So the backward propagation proves as expected that the types of the inputs are $\langle \texttt{x0}\rangle$ and $\langle \texttt{xu}\rangle$ respectively. In this example the fact that $a_0 = 0$ was not useful to infer more information, but if we change the operator from `^` to `&`, this information provides the type $\langle \texttt{x}\rangle$ 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, described fully in [5].

The Dataflow Analysis. We maintain for each value two types: the *best* type and the *current* type. The *best* type is initialized conservatively ($\perp$) and moves up in the lattice after each pass. The analysis works by alternating 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 to $\top$, 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*.

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

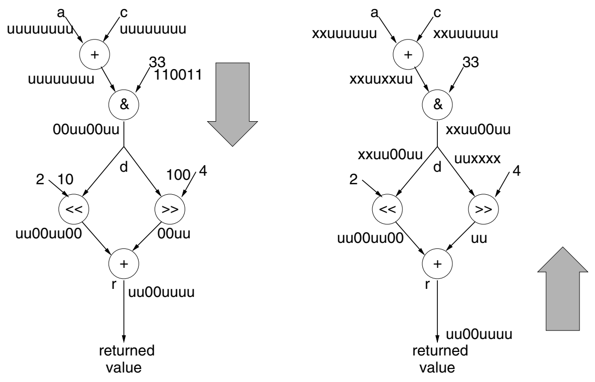

Fig. 2. *A C function and the associated data-flow graph. The types inferred by forward (backward) propagation are shown in the left (righ) figure. We assume that a* `char` *has 8 bits.*

2.1 Example

We illustrate the algorithm on the code in Figure 2.[1] The algorithm begins with the forward pass and examines the first statement. The sources for the first statement are parameters which are defined outside the procedure and thus are set to be all *don't knows*, i.e. every bit is significant. `c+a` from Figure 2 must be computed on 9 bits. The result is truncated to 8 bits of precision because of the definition of `char` in the underlying language implementation. 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 generates 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 $\cap$ of these two strings. The middle two bits of `d` have been found to be 0 by forward propagation, and they are not changed.

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.

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

Table 1. *Percent reduction in bitwidth for programs in MediaBench (left) and SpecINT95 (right). We are only counting the most significant bytes. The column labeled "bitv" indicates that only the BitValue was run, "ind" indicates that only loop induction-variable analysis was performed, and "both" indicates both analyses were performed. The results are rounded down; a zero means "less than one percent". We were unable to profile gcc.*

	Static %			Dynamic %		
Benchmark	ind	bitv	both	ind	bitv	both
adpcm_e	0	19	19	0	19	19
adpcm_d	0	19	19	0	24	24
g721_Q_d	1	32	33	4	26	31
g721_Q_e	1	32	33	4	25	29
gsm_e	1	30	31	7	7	14
gsm_d	1	30	31	4	24	24
epic_e	0	5	5	0	0	0
epic_d	0	3	3	0	4	4
mpeg2_e	0	12	13	24	4	28
mpeg2_d	0	9	10	1	7	8
jpeg_e	0	4	4	1	7	9
jpeg_d	0	4	4	0	11	11
pegwit_e	0	14	15	0	13	13
pegwit_d	0	14	15	0	16	16
mesa	0	5	5	0	5	5

	Static %			Dynamic %		
Benchmark	ind	bitv	both	ind	bitv	both
124.m88ksim	1	22	22	1	19	20
129.compres	2	11	13	0	11	12
099.go	0	6	7	0	2	2
130.li	0	14	14	0	12	12
132.ijpeg	0	5	5	1	10	11
134.perl	0	11	11	0	8	8
147.vortex	0	6	6	0	5	5
126.gcc	0	19	19	*	*	*

3 Experiments with a C Compiler

We evaluate our algorithm implemented in SUIF [15] on C programs from MediaBench [8] and SpecINT95 [12]. BitValue is implemented as a work-list based dataflow algorithm starting from def-use chains [14]. Both def-use and BitValue are local analyses. Information from alias analysis or an interprocedural BitValue analysis would improve our results, at a cost of greater compilation time.

3.1 Evaluation

In this section we compare the merits of induction variable analysis, BitValue, and the interaction between them. Induction variable analysis has been used in [13] to compute ranges of values for each variable which is used to reduce the number of necessary bits. We have used a simplified form of this analysis to analyze FORTRAN-style `for` loops. We only detect values which depend linearly on the loop index.

We ran three experiments for each benchmark: the induction-variable analysis only, BitValue only, and both. When we ran both analyses, we first ran the induction-variable analysis, and we fed the bounds derived by it into the initial information for BitValue.

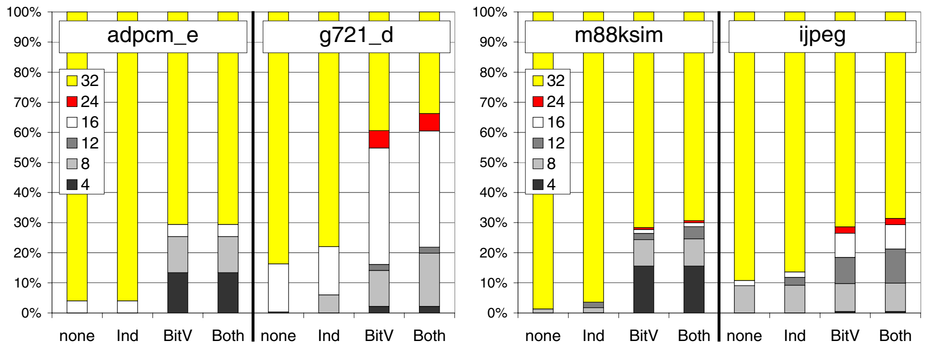

Fig. 3. *Percentage breakdown of widths from some programs (dynamic counts).*

Depending on the hardware model which exploits the narrow bitwidths, not every constant or *don't care* bit can be eliminated. For instance, a constant bit in the middle of a byte cannot be discarded when using subword parallelism. To account for this, the data in Table 1 counts only the most significant bytes as useless. For example, in a 16-bit data item with inferred type ⟨`x001u001 xxxuuuxx`⟩ we count *no* saved bytes because there is a useful bit in each of the bytes. These results underestimate the performance of the algorithm but apply to a wider range of architectures. If we count all the bit savings, we obtain on average an additional 6% reduction.

Most often BitValue and the induction-variable analysis complement (or even reinforce) each other: there are benchmarks (e.g., jpeg_e) where the "both" count surpasses the sum of the two other counts.

In Figure 3 we show the histograms of the data sizes operated upon for some selected benchmarks. The value sizes are rounded up. For each program we present four histograms: one for the original program (with no analysis), one for the induction variable analysis alone, one for BitValue analysis alone, and one for both analyses (induction followed by BitValue).

For example, we can interpret the graphs for `adpcm_e` in the following way: the first bar says that about 5% of the values in the original program are 16-bit or less. The fourth bar shows that using BitValue we discover that 16 bits are actually enough for about 30% of the values in the program.

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, addition and multiplication by small constants are the most powerful; (2) the propagation of cast information through the backward analysis; (3) array element index computations.

A preliminary evaluation of the benefits of discovering narrow values shows that the analysis is important in the context of reconfigurable functional units (RFUs). We used as a target architecture a VLIW processor augmented with an PipeRench-like [7] RFU on the data path. For example, compiling g721_e for the processor+RFU combination without the analysis yields an 18% reduction in the running time. If we optimize the portion mapped to the RFU using BitValue we obtain a 26% reduction in running time.

3.2 Practical Issues

Our implementation of BitValue is fast and scales linearly in practice with program size. The space complexity is linear. We analyze on average 900 lines/second on a PIII @750Mhz, with an untuned implementation.

An interesting side-effect of our analysis is that it gives a portable high-level method for specifing widths: by using a masking operation we 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.

Our current implementation runs the induction variable analysis only once and BitValue afterwards. Improvements can be obtained by iterating these analyses until a fixed-point is reached. Future work will investigate the possible gains.

4 Experiments with a Reconfigurable Hardware Compiler

In this section we evaluate the BitValue algorithm as it is used in the DIL compiler [4] which we developed for 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.

Because of this 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-bitsand then we run the algorithm to determine the reduction in size.

Table 2 shows the amount of hardware required to implement kernels compiled with the DIL compiler. Note that the impact of the analysis is significant: it can decrease the silicon real-estate (and implicitly, decrease the power consumption and decrease the latency of the computation) with a factor between 3 and 20.

5 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 30% of the widths are 16 bits or less for any input data. They suggest hardware techniques for creating instructions which operate on narrow widths on the fly. The work of Bondalapati and Prasanna is similar, looking

Table 2. *The size of the circuits in bit-operations/8, for two circuit versions: one where all values are 32-bit and one with variable sizes. The percent column shows the remaining size of the circuit after optimizations (the smaller, the better).*

Program	Description	Original	Final	%
cordic	12 stage implementation of Cordic vector rotations	1507	332	23
encoder	8-bit Huffman encoder with the code table hardwired	2286	578	26
dct	1-D 8-point Discrete Cosine Transform	366	94	26
fir	FIR filter with 20 taps and 8-bit coefficients	320	123	39
idea	Complete 8 round key-specific International Data Encryption Algorithm	2074	576	28
nqueens	Evaluator for the n-queens problem on an 8x8 board	144	7	5
over	Porter-Duff “over” operator	280	49	18
popcnt	Count the number of “1” bits in a 16-bit word.	96	5	6

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. Our algorithm has worst-case quadratic complexity. We compared our algorithm to the Synopsis Synplify compiler, a commercial CAD tool, using the DCT benchmark from Section 4. 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 know 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 [13]. These techniques are orthogonal to ours. Our analysis would very likely combine well with this technique, because the results of one could be used to seed the starting point of the other one, in the same way we handle induction variables.

6 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).

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 three- and twenty-fold. Finally, using our algorithm we were able to increase the simulated performance of several MediaBench programs by more than 20% when run on a CPU with a reconfigurable functional unit. 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.

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. M. Budiu and S.C. Goldstein. BitValue — Detecting and Exploiting Narrow Bitwidth Computations. Technical Report CMU-CS-00-141, Carnegie Mellon University, June 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. http://www.specbench.org/osg/cpu95/.
13. 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.
14. 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.
15. 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.