Bitwidth Analysis with Application to Silicon Compilation

Mark Stephenson, Jonathan Babb, and Saman Amarasinghe
Massachusetts Institute of Technology
Laboratory For Computer Science
Cambridge, MA 02139
{mstephen,saman}@lcs.mit.edu, jbabb@ee.princeton.edu

Abstract

This paper introduces *Bitwise*, a compiler that minimizes the bitwidth — the number of bits used to represent each operand — for both integers and pointers in a program. By propagating static information both forward and backward in the program dataflow graph, Bitwise frees the programmer from declaring bitwidth invariants in cases where the compiler can determine bitwidths automatically. We find a rich opportunity for bitwidth reduction in modern multimedia and streaming application workloads. For new architectures that support sub-word quantities, we expect that our bitwidth reductions will save power and increase processor performance.

This paper also applies our analysis to silicon compilation — the translation of programs into custom hardware — to realize the full benefits of bitwidth reduction. We describe our integration of Bitwise with the DeepC Silicon Compiler. By taking advantage of bitwidth information during architectural synthesis, we reduce silicon real estate by 15% - 86%, improve clock speed by 3% - 249%, and reduce power by 46% - 73%. The next era of general purpose and reconfigurable architectures should strive to capture a portion of these gains.

1 Introduction

The pioneers of the computing revolution described in Steven Levy's book *Hackers*, competed to make the best use of every precious architectural resource. They hand-tuned each program statement and operand. In contrast, today's programmers pay little attention to small details such as the bitwidth (*e.g.*, 8, 16, 32) of data types used in their programs. For instance, in the C programming language, it is common to use a 32-bit integer data type to represent a single Boolean variable! We could dismiss this shift in emphasis as a consequence of abundant computing resources and expensive programmer time. However, there is another historical reason — as processor architectures have evolved, the use of smaller operands eventually has provided no performance gains. Datapaths became wider, but the processor's entire data path was exercised regardless of operand size. The additional overhead of packing and unpacking words — now only to save space in memory — even reduced performance.

1.1 A New Era: Software-Exposed Bits

There are three new compilation targets for high-level languages that are re-invigorating the need to conserve bits. Each of these architectures exposes subword control. The first is the renovation of SIMD architectures for multimedia workloads. These architecture include Intel's MultiMedia eXtension (MMX), and Motorola's Altivec. For example, in Altivec, data paths are used to operate on 8, 16, 32, or 64 bit quantities.

The second class of compilation targets comprises fine-grain substrates such as reconfigurable architectures — including Field Programmable Gate Arrays (FPGAs) — and custom hardware, such as ASIC and standard cell designs. In both cases, architectural synthesis is required to support high-level languages. There has been a recent surge of both industrial and academic interest in developing new reconfigurable architectures. And for custom silicon, the emphasis on high-level compilation has been accelerated by a consortium of over 50 companies who have recently formed the Open System C Initiative [13] to focus on the standardization of high-level compilation.

The third class of compilation targets consists of embedded systems which can effectively *turn off* bit slices [5]. The static information determined at compile time can be used to control which portions of a datapath are on or off during program execution. Alternatively, for more traditional architectures this same information can be used to optimize power consumption (without actually running the program) by predicting which bits on a datapath will change over time.

But there are no available commercial compilers that can effectively target any of these new architectures. So programmers have been forced to revert back to writing low-level code. MMX libraries are written in assembly in order to expose the most sub-word parallelism. In Verilog and VHDL hardware description languages, the burden of bitwidth specification is on the programmer. To compete in the marketplace, designers must choose the minimum operand bitwidth for smaller and faster and more energy efficient circuits. Unfortunately, explicitly choosing the smallest data size for each operand is not only tedious, but also error prone. These programs are less malleable since a simple change may require hand propagation of bitwidth information across a large segment of the program. Furthermore, some of the bitwidth information may be dependent on a particular architecture or implementation technology, making the programs less portable.

1.2 Automating Bitwidth Specification

Automatic bitwidth analysis relieves the programmer of the burden of identifying and specifying derivable bitwidth information. The programmer can work at a higher level of abstraction. Even if the programmer explicitly specifies operand sizes in languages which allow it, bitwidth analysis can still be very valuable. For example, bitwidth analysis can be used to verify that specified operand sizes do not violate program invariants – *e.g.*, array bounds. Or bitwidth analysis can be used to change a single variable's bitwidth throughout the life of the variable.

1.3 The Bitwise Compiler

Bitwise minimizes the bitwidth required for each static operation and each static assignment of the program. The scope of Bitwise includes fixed-point arithmetic, bit manipulation and boolean operations. It uses additional sources of information such as type casts, array bounds, and loop iteration counts to refine the bitwidth information gathered. We have implemented Bitwise using the SUIF compiler infrastructure [17].

In many cases, Bitwise is able to analyze the bitwidth information as accurately as the bitwidth information gathered from run-time profiles. On average we reduce the size of program scalars by 12% - 80% and program arrays by up to 93%.

1.4 Application to Silicon Compilation

In the paper we will focus on the application of bitwidth analysis to silicon compilation. We have integrated Bitwise with the *DeepC Silicon Compiler*. The compiler produces hardware netlists from input programs written in C and FORTRAN. We report end-to-end performance results for this system both with and without the Bitwise optimizations. The results show how well the analysis works in the context of a real system. Our experiments show Bitwise favorably impacts area, speed, and power of the resulting circuits.

1.5 Contributions

We summarize this paper's contributions as follows.

- We formulate bitwidth analysis as a value range propagation problem.
- We introduce a suite of bitwidth extraction techniques that seamlessly perform bi-directional propagation.
- We formulate an algorithm to accurately find bitwidth information in the presence of loops by calculating closed-form solutions.
- We implement the analysis and demonstrate that the compile-time analysis can approach the accuracy of run-time profiling.
- We incorporate the analysis in a silicon compiler and demonstrate that bitwidth analysis impacts area, speed, and power consumption of a synthesized circuit.

1.6 Organization

The rest of the paper is organized as follows. Section 2 defines the bitwidth analysis problem. Bitwise's implementation and our algorithms are described in Section 3. Section 4 provides empirical evidence of the success of Bitwise. Next, Section 5 describes the DeepC Silicon Compiler and Section 6 discusses the impact of bitwidth analysis to silicon compilation. Finally, we present related work in Section 7 and conclude in Section 8.

2 Bitwidth Analysis

Bitwidth analysis attempts to discover the smallest types for each static variable assignment in a program while retaining program correctness. A static variable assignment is defined as an assignment in SSA form.

Library calls, I/O routines, and loops make static bitwidth analysis challenging. In the presence of these constructs, we may have to make conservative assumptions about an operand's bitwidth. Nevertheless, with careful static analysis, it is possible to infer bitwidth information.

Structures such as arrays and conditional statements provide us with valuable bitwidth information. For instance, we can use the bounds of an array to set an index variable's maximum bitwidth. Other program constructs such as AND-masks, divides, right shifts, type promotions, and Boolean operations are also invaluable for reducing bitwidths.

The C code fragment in Figure 2 exhibits several such constructs. This code — which is an excerpt of one of the benchmarks presented in this paper (adpcm) — is typical of tomorrow's important multimedia applications. Each line of code in the figure is annotated with a line number to facilitate the discussion that follows.

Assume that we don't know the precise value of `delta`. Because it is used as an index variable in line (1), if we assume that this is a legal program, we know that its value is confined by the base and bounds of `indexTable`. Though we still don't know `delta`'s precise value, by restricting the range of values that it can assume, we effectively reduce the number of bits needed to represent it. In a similar fashion, the code on lines (2) and (3) ensure that `index`'s value is restricted to be between 0 and 88.

The and-mask in line (7) ensures that `outputbuffer`'s value is no greater than `0xf0`. We can propagate this information to infer that the assignment to `*outp` in line (9) is no greater than `0xff` (`0x0f | 0xf0`).

Finally, we know that `bufferstep`'s value is either *true* or *false* after the assignment in line (11) because it is the result of the Boolean *not* (`!`) operation.

3 Bitwise Implementation

We next introduce the *Bitwise* compiler, a new set of compiler passes that perform bitwidth analysis. We begin by describing the compiler's infrastructure. A description of the algorithms follows in Section 3.2.

3.1 Infrastructure

The *Bitwise* compiler uses SSA as its intermediate form. The compiler performs a *numerical* data flow analysis. Fortunately, we do not need the more complex symbolic analysis because we are solving for absolute numerical bitwidths.

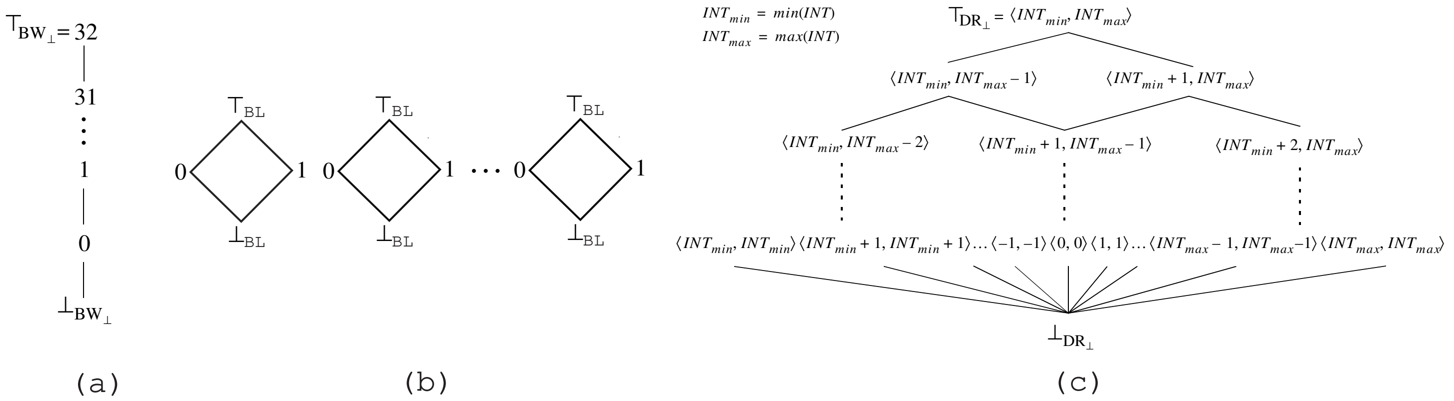

Figure 1: Three alternatives data structures for bitwidth analysis. The lattice in (a) represents the number of bits needed to represent a variable. The lattice in (b) represents a vector of bits that can be assigned to a variable, and the lattice in (c) represents the range of values that can be assigned to a variable.

```
index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;
step = stepsizeTable[index];

if ( bufferstep ) {
  outputbuffer = (delta << 4) & 0xf0;
} else {
  *outp++ = (delta & 0x0f) | outputbuffer;
}
bufferstep = !bufferstep;
```

Figure 2: Sample C code used to illustrate the fundamentals of the analysis. This code fragment was taken from the loop of `adpcm_coder` in the adpcm multimedia benchmark.

This section describes the choice of data-structures for propagating numerical information in our analysis. We consider three candidate data-structures. Figure 1 visually depicts each *lattice*, a formal ordering of their internal structure.

Propagating the bitwidth of each variable: Figure 1(a) is the most straightforward implementation, and has been described by Scott Ananian [1]. While this representation permits an easy implementation, it does not yield accurate results on arithmetic operations. When applying the lattice's transfer function, incrementing an 8-bit number always produces a 9-bit resultant, even though it may likely only need 8-bits. In addition, only the most significant bits of a variable are candidates for bit-elimination.

Maintaining a bit vector for each variable: Figure 1(b) is a more complex representation, requiring the composition of several smaller bit-lattices. Although this lattice allows elimination of arbitrary bits from a variable's representation, it does not support precise arithmetic analysis. As an example of eliminating arbitrary bits, consider a particular variable that is assigned the values from the set, $\{010_2, 100_2, 110_2\}$. After analysis, the variable's bit-vector will be $[\top\top 0]$, indicating that we can eliminate the least significant bit. Like the first data structure, the arithmetic is imprecise because the analysis must still conservative assume that every addition results in a carry.

Propagating data-ranges: Figure 1(c) is the final lattice we considered. This lattice is also the implementation chosen in the compiler. A data-range is a single connected subrange of the integers from a lower bound to an upper bound (*e.g.*, [1..100] or [-50..50]). Thus a data-range keeps track of a variable's lower and upper bounds. Because only a single range is used to represent all possible values for a variable, this representation does not permit the elimination of low-order bits. However, it does allows us to operate on arithmetic expressions precisely. Technically, this representation maps bitwidth analysis to the more general *value range propagation problem.* Value range propagation is known to be useful in value prediction, branch prediction, constant propagation, procedure cloning, and program verification [14].

For the *Bitwise* compiler we choose to propagate data-ranges, not only because of their generality, but also because most important applications use arithmetic and will benefit from their exact precision. The lattice in Figure 1(c) is a *lifted lattice* in that it includes a bottom element. The value $\perp_{DR_\perp}$ represents a value that have not yet been initialized. Additionally, note that the value $\top_{DR_\perp}$, a part of the lattice, represents a values that cannot be statically determined. Finally, unlike a regular set union, we define data range union ($\sqcup$) to be the union over the single connected subrange of the integers where $\langle a_l, a_h \rangle \sqcup \langle b_l, b_h \rangle = \langle min(a_l, b_l), max(a_l, b_h) \rangle$. We define data range intersection ($\sqcap$) to be the set of all integers in both subranges where $\langle a_l, a_h \rangle \sqcap \langle b_l, b_h \rangle = \langle max(a_l, b_l), min(a_h, b_h) \rangle$.

3.2 Data-Range Propagation

Data-ranges can be propagated both *forward* and *backward* over the control flow graph. Figure 4 shows a subset of the transfer functions for propagation. The forward propagated values in the figure are subscripted with a down arrow ($\downarrow$), and the backward propagated values with an up arrow ($\uparrow$). In general the transfer functions take as input either one or two data-ranges and return a single data-range.

We begin with a discussion of forward propagation. An *SSA graph* is the control flow graph in SSA form. Initially, all of the variables in the SSA graph are initialized to $\top_{DR_\perp}$. Informally, forward propagation traverses the SSA graph in breadth-first order, applying the transfer functions for forward propagation. Because forward propagation is so well known, the details are omitted here.

Forward propagation allows us to identify a significant number of unused bits, sometime achieving the optimal result. However, additional minimization can be achieved by integrating *backward propagation*. For example, when we find a data-range that has stepped outside of known boundaries, we can back propagate this new reduced data-range to instructions that have already used its deprecated value to compute their results. Beginning at the node where the boundary violation is found, we propagate the reduced data-range in a reverse breadth-first order. Backward propagation halts when either the graph's entry node is reached, or when a fixed point is reached.

To further elaborate, consider the pedagogical SSA graph shown in Figure 3. Forward and backward propagation steps have been annotated on the graph. The numbers to the right of the figure list each step. The step numbers in black represent the backward propagation of data-ranges. Without this backward propagation we would arrive at the following data-ranges:

$\texttt{a0} = \langle INT_{min}, INT_{max} \rangle$
$\texttt{a1} = \langle INT_{min}, INT_{max} \rangle$
$\texttt{a2} = \langle INT_{min} + 1, 0 \rangle$
$\texttt{a3} = \langle INT_{min} + 1, INT_{max} \rangle$
$\texttt{c0} = \langle 0, INT_{max} \rangle$

Let us assume we know that the length of the array, `array`, is 10 from its program declaration. We can now substantially reduce the data-ranges of these variables with backward propagation. We use `array`'s bound information to clamp `a3`'s data-range to $\langle 0, 10 \rangle$. We then propagate this value backward in reverse breadth-first order using the transfer functions for backward propagation. In our example, propagating `a3`'s new value backward yields the following new data-ranges:

$\texttt{a0} = \langle -2, 9 \rangle$
$\texttt{a1} = \langle -1, 10 \rangle$
$\texttt{a2} = \langle 0, 0 \rangle$

Reverse propagation can halt after `a0`'s range is determined (step 14). Because `c0` uses the results of a variable that has changed, we have to traverse the graph in the forward direction again. After we confine `c0`'s data-range to $\langle 0, 10 \rangle$ we will have reached a fixed point and the analysis is complete.

In this example we see that data-range propagation *subsumes* constant propagation; we can replace all occurrences of `a2` with the constant value 0.

Before considering loops, we can informally reason about termination. After the application of each transfer function, a variable's data-range will shrink by some amount. In theory, if we repeatly apply the rules, we will eventually reach a fixed point.

However, the height of the lattice may not make this a practical solution[1].

Non-termination was not a problem for our benchmarks. However, when it is a problem, there is an easy solution. Because successive iterations reduce the range of numbers a variable can represent— and thus the bitwidth of the variable— we can stop iterating after a user-defined number of iterations. At that point the analyzer will have computed a reduced but potentially sub-optimal bound for the variable's bitwidth.

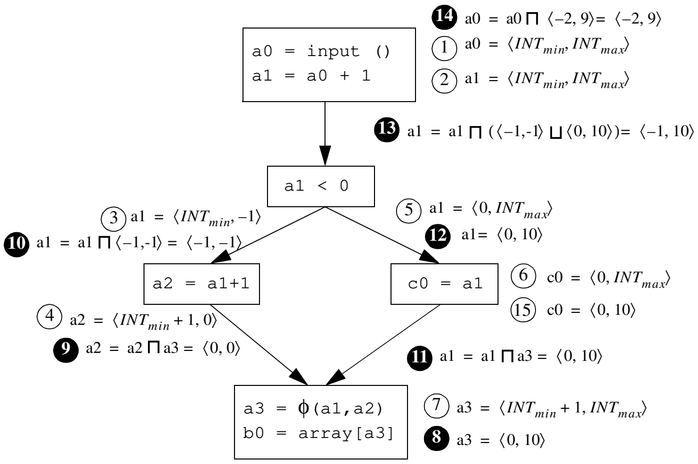

Figure 3: Forward and Backward Data-Range Propagation. Application of forward propagation rules are shown to the right of the figure in white, while backward propagation rules are shown in black. We use `array`'s bounds information to tighten the bounds on some of the variables.

3.3 Loops

Optimization of loop instructions is crucial — they usually comprise the bulk of dynamic instructions. Traditionally, data flow analysis techniques iterate over back edges in the graph until a fixed point is reached. But in the presence of even simple loop-carried expressions, this technique will saturate bitwidths. That is, because the method does not take into account any static knowledge of loop bounds, related variables will end up being 32-bits for a typical 32-bit integer declarations.

But many important applications use loop-carried arithmetic expressions. A new approach is required; in our case, we attempt to find closed-form solutions. When we can find a closed-form solution, the height of the lattice becomes irrelevant!

3.3.1 Loop Extension to SSA Form

In order to search for closed-form solutions, we must first identify loops. We ease loop identification in SSA form by converting all ϕ-functions that occur in loop headers to μ-functions [7]. This extension is assumed for the remainder of the paper.

3.3.2 Loop Analysis

To find the closed form solution to loop-carried expressions, we use the techniques introduced by Gerlek et. al.[7]. These techniques allow us to identify and classify *sequences* in loops. A *sequence* is a group of instructions that are mutually dependent on all of the other instructions in the same group. In other words, a sequence is a strongly connected component (SCC) of the program's dependence graph. We can examine the instructions of the sequence to try and find a closed form solution to the sequence. The algorithm for detecting and classifying sequences is shown in Figure 5.

We next define the function `sequence_type` of the `classify_sequence` procedure. We can create a partial or-

[1] If we assume that data-ranges are composed of 32-bit integers, we can derive the height of the data-range lattice as follows:

$$DR = \{< i, j > | i, j \in INT, i \leq j\}$$

$$|DR| = \sum_{i=1}^{|INT|} |INT| - i + 1 = \frac{|INT| \cdot (|INT| + 1)}{2}$$

$$|DR| = 2^{64}$$

	Inputs	Statement	Transfer functions
(a)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b + c $\downarrow$	$b_\uparrow = b_\downarrow \sqcap \langle a_l - c_h, a_h - c_l \rangle$ $c_\uparrow = c_\downarrow \sqcap \langle a_l - b_h, a_h - b_l \rangle$ $a_\downarrow = a_\uparrow \sqcap \langle b_l + c_l, b_h + c_h \rangle$
(b)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b - c $\downarrow$	$b_\uparrow = b_\downarrow \sqcap \langle a_l + c_l, a_h + c_h \rangle$ $c_\uparrow = c_\downarrow \sqcap \langle a_l + b_l, a_h + b_h \rangle$ $a_\downarrow = a_\uparrow \sqcap \langle b_l - c_h, b_h - c_l \rangle$
(c)	$b_\downarrow = \langle b_l, b_h \rangle$ $c_\downarrow = \langle c_l, c_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$\downarrow$ a = b & c $\downarrow$	$b_\uparrow = b_\downarrow$ $c_\uparrow = c_\downarrow$ $a_\downarrow = a_\uparrow \sqcap \left\langle -2^{n-1}, 2^{n-1} - 1 \right\rangle$, where $n = min(bitwidth(b_\downarrow), bitwidth(c_\downarrow))$
(d)	$b_\downarrow = \langle b_l, b_h \rangle$ $a_\uparrow = \langle a_l, a_h \rangle$	$TypeA : \langle A_L, A_H \rangle$ $TypeB : \langle B_L, B_H \rangle$ typeA a , typeB b $\downarrow$ a = b $\downarrow$	$b_\uparrow = b_\downarrow \sqcap a_\uparrow \sqcap \langle A_L, A_H \rangle$ $a_\downarrow = a_\uparrow \sqcap b_\downarrow \sqcap \langle B_L, B_H \rangle$
(e)	$x_\downarrow = \langle a_l, a_h \rangle$	$\downarrow$ $\{x_l \leq$ x $\leq x_h\}$ $\downarrow$	$x_\uparrow = x_\downarrow \sqcup \langle x_l, x_h \rangle$
(f)	${x^b}_\downarrow = \langle b_l, b_h \rangle$ ${x^c}_\downarrow = \langle c_l, c_h \rangle$ ${x^a}_\uparrow = \langle a_l, a_h \rangle$	$x^b \searrow \quad \swarrow x^c$ $x^a = \phi(x^b, x^c)$ $\downarrow$	${x^b}_\uparrow = {x^b}_\downarrow \sqcap {x^a}_\uparrow$ ${x^c}_\uparrow = {x^c}_\downarrow \sqcap {x^a}_\uparrow$ ${x^a}_\downarrow = {x^a}_\uparrow \sqcap ({x^b}_\downarrow \sqcup {x^c}_\downarrow)$
(g)	${x^a}_\downarrow = \langle a_l, a_h \rangle$ $y_\downarrow = \langle y_l, y_h \rangle$ ${x^b}_\uparrow = \langle b_l, b_h \rangle$ ${x^c}_\uparrow = \langle c_l, c_h \rangle$	x^a $\downarrow$ $x^a < y$ $\swarrow \quad \searrow$ $x^b \quad x^c$	${x^a}_\uparrow = {x^a}_\downarrow \sqcap ({x^b}_\uparrow \sqcup {x^c}_\uparrow)$ ${x^b}_\downarrow = {x^a}_\downarrow \sqcap {x^b}_\uparrow \sqcap \langle a_l, y_h - 1 \rangle$ ${x^c}_\downarrow = {x^a}_\downarrow \sqcap {x^c}_\uparrow \sqcap \langle y_l, a_h \rangle$

Figure 4: A selected subset of transfer functions for bi-directional data-range propagation. Equations on the left are inputs to the transfer functions on the right. The variables in the figure are subscripted with the direction in which they were computed. The transfer function in (a) adds two data-ranges, and (b) subtracts two data-ranges. The AND-masking operation in (c) returns a data-range corresponding to the smallest of its two inputs. It makes use of the *bitwidth* function which returns the number of bits needed to represent the data-range. The type-casting operation shown in (d) constricts the propagated data-range to be at most, the maximum range that can be represented by its input type. The function in (e) is applied when we know a value must be within a specified range. For instance, this rule is applied to limit the data-range of a variable that is indexing into a static array. Rules (f) and (g) are applied at confluence points.

```
Purpose:
    ; This procedure detects sequences in the graph
    ; of ssa nodes passed in. If a commonly occuring
    ; sequence is found, its closed form solution is
    ; computed.
Given:
    ; graph_nodes : A graph of ssa nodes that comprise a loop.
procedure find_closed_form_solutions (graph_nodes)
    entry : ssa_node
    tripcount : integer
    components : graphof listof instruction
    seq1, seq2, sequence : listof instruction

    ;; The algorithm assumes that the head of the loop is
    ;; annotated with the loop's tripcount. If the loop is
    ;; a while-type loop, the tripcount will be infinity.
    entry ← entry node of graph_nodes
    tripcount ← entry.tripcount

    foreach instr ∈ entry do
        if instr is of type μ-funtion then
            instr.tripcount ← tripcount

    ;; Use Tarjan's algorithm to find connected components
    components ← find_SCCs (graph_nodes)
    foreach seq1 ∈ components do
        foreach seq2 ∈ components do
            if seq1 is dependent on seq2 then
                add vertex in components from seq1 to seq2

    foreach sequence ∈ components do
        if sequence has no outgoing vertices then
            classify_sequence (sequence)
            components ← components −graph sequence
end procedure

Purpose:
    ; This procedure classifies the sequence that is passed in
    ; according to its composition of instructions.
Given:
    ; Inputs : A list of instructions.
procedure classify_sequence (sequence)
    seqtype : oneof {boolean, linear, polynomial, geometric, top}
    tripcount : integer
    initial : datarange
    final : datarange
    value : datarange

    if sizeof sequence = 1 then
        instr ← instruction in sequence
        instr.destination ← evaluate_instruction (instr)
    else
        ;; Determine the type of sequence that we dealing with.
        seqtype ← sequence_type (sequence)

        if seqtype = boolean then
            foreach instr ∈ sequence do
                instr.destination ← ⟨0, 1⟩
        elseif seqtype = linear
            instr ← μ-function in sequence
            tripcount ← instr.tripcount
            initial ← evaluate_instruction (instr)
            foreach instr ∈ sequence do
                final ← evaluate_instruction (instr)
                instr.detination ← final
            growth ← (final −DR initial) ∗DR tripcount
            foreach instr ∈ sequence do
                instr.destination ← instr.detination +DR growth
        elseif ...
            .
            .
            .
        elseif seqtype = top_sequence
            foreach instr ∈ sequence do
                instr.destination ← ⊤DR
end procedure
```

Figure 5: Pseudocode for the algorithms that detect, classify, and compute closed form solutions of commonly occuring sequences. The function `find_closed_form_solutions` detects sequences in code and calls `classify_sequence` to classify the sequences and compute the closed form solutions.

der on the types of expressions we wish to identify. The *Expression* lattice (Figure 6) orders various expressions according to set containment. The top of the lattice represents an undetermined expression, while the bottom of the lattice represents all possible expressions. Linear sequences represent induction variables in loop bodies. Polynomial induction sequences represent a composition of linear sequences. Likewise geometric sequences are composed of polynomial sequences and linear sequences.

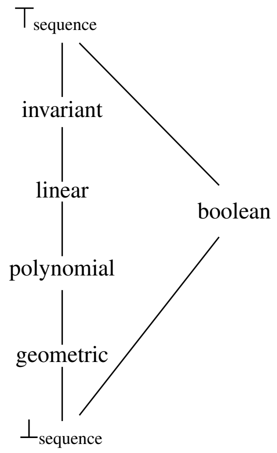

Figure 6: A lattice that orders sequences according to set containment.

For each instruction type in the source language, we create transfer functions that operate on the lattice. A transfer function is implemented as a table that is indexed by the expression types of its source operands. The destination operand is then tagged with the expression type dictated by the transfer function. See [7] for a more detailed explanation.

We extend the work done by Gerlek et. al. [7] by identifying boolean sequences. These sequences are not only easy to find, but they also allow us to represent each boolean static assignment with only one bit.

After we have determined the type of each expression in a sequence, we can classify it based on the types of its expressions and its composition of ϕ- and $\mu-$functions. For instance, *boolean* sequences can contain any number of ϕ- or μ-functions, but can only contain *boolean* sequences.

Once we have determined the type of sequence the component represents, we use a *solver* to compute the sequence's closed form solution. Each type of sequence has its own solver that takes as input the sequence and the initial values of the variables. As an example, consider the simple code fragment below:

```
a = 0;
for (i = 0; i < 10; i++) {
  a = a + 5;
}
```

The analyzer first converts the code to SSA form, then it ascertains the symbolic tripcount of the loop, where tripcount is defined as the number of times the loop is iterated. In this case, the loop's tripcount is determined to be 10. Next the analyzer, finds all of the strongly connected components in the loop's body; these components represent the sequences. For this example, the only component is shown in Figure 7.

Sequence	*Iteration*1	*Iteration*2
$\texttt{a1} = \mu(\texttt{a0}, \texttt{a2})$	$\langle 0, 0\rangle$	$\langle 0, 50\rangle$
$\texttt{a2} = \texttt{a1} + 5$	$\langle 5, 5\rangle$	$\langle 5, 50\rangle$

Figure 7: An example that shows the detection and computation of a linear sequence. The sequence is shown on the left, and the computation of the sequence is shown on the right.

Because the sequence contains only a single μ-function and a linear-type expression, we use the solver for linear components. The solving process is traced to the right of the sequence in the figure. In the first iteration, we compute the growth factor of the sequence. The resulting data-ranges are shown in the column labeled, *Iteration*1. Note that when a solver evaluates a μ-function, during the first iteration it only uses the operand that is defined outside the loop body. Subsequent iterations use both operands.

We can then multiply the growth factor by the tripcount to determine the final result of the sequence. The second iteration over the loop simply adds the final value of the sequence to the value computed in the first iteration.

In many cases, using this technique obviates the need for finding a fixed point. However, when we are unable to identify a sequence, we can set all of the static assignments in the sequence to the maximum data-range available ($\top_{DR_\perp}$). Alternatively we can iterate over the sequence until a fixed point is reached or until we reach a user-defined maximum number of iterations. After the maximum number of iterations is reached, the destination operands in the sequence are set to $\top_{DR_\perp}$.

3.4 Arrays

In traditional SSA form, pointers and arrays are not renamed. Special extensions to SSA form have been proposed which provide element-level data flow information for arrays [11]. While such extensions to SSA form can potentially provide more accurate data-range information, for bitwidth analysis it is actually more convenient to treat arrays as scalars; this analysis is inexpensive from a complexity standpoint, and when compiling to silicon this analysis accurately determines the data bus size for embedded RAMs.

Wherever an array is modified we have to insert a new ϕ-function to merge the array's old data-range with the new data-range. One drawback of this method is that a ϕ-function is required for every array assignment, increasing the size of the code. However, def-use chains are still inherent in the intermediate representation which simplifies the analysis.

3.5 Pointers

Using pointer analysis such as Radu Rugina's SPAN package [15], we can determine the sets of variables — commonly referred to as *location sets* — a pointer *may* or *must* reference. Such an analysis package tags all memory references with location set information.

3.5.1 Example and Discussion

To simplify this discussion, we will distinguish between *reference location sets*, and *modify location sets*: a reference location set is a location set annotation that occurs on the right hand side of an expression, whereas a modify location set occurs on the left hand side of an expression.

Benchmark	Type	Source	Lines	Description
adpcm	Multimedia	UTdsp	195	Audio Compress
bubblesort	Scientific	Raw	62	Bubble Sort
convolve	Multimedia	MIT	74	Convolution
histogram	Multimedia	UTdsp	115	Histogram
intfir	Multimedia	UTdsp	64	Integer FIR
intmatmul	Scientific	Raw	78	Int. Matrix Mult.
jacobi	Scientific	Raw	84	Jacobi Relation
life	Automata	Raw	150	Game of Life
median	Multimedia	UTdsp	86	Median Filter
mpegcorr	Multimedia	Berkeley	144	MPEG-3 Kernel
newlife	Automata	MIT	119	New Game of Life
parity	Multimedia	MIT	54	Parity Function
pmatch	Multimedia	MIT	63	Pattern Matching
sor	Scientific	MIT	60	5-point Stencil
sha	Encryption	MIT	638	Secure Hash
softfloat	Emulation	Berkeley	1815	Floating Point

Table 1: Benchmark characteristics

As an example, consider the following C memory instruction, assuming that `p0` is a pointer that can point to variable `a0` or `b0`, and that `q0` is a pointer that can only point to variable `b0`:

```
*p0 = *q0 + 1
```

The location set that the instruction may modify is $\{a0, b0\}$, and the location set that the instruction must reference is $\{b0\}$. Since there is only one variable in the instruction's reference location set, it *must* reference `b0`. Since there are two variables in the modify location set, either `a0` or `b0` *may* be modified.

Keeping the SSA guarantee that there is one unique assignment associated with each variable, we have to rename `a0` and `b0` in the instruction's modify location set. Furthermore, since it is not certain that either variable will be modified, a ϕ-function has to be inserted for each variable in the modify location set to merge the previous version of the variable with the renamed version:

$$\{\texttt{a1}, \texttt{b1}\} = \{\texttt{b0}\} + 1$$
$$\texttt{a2} = \phi(\texttt{a0}, \texttt{a1})$$
$$\texttt{b2} = \phi(\texttt{b0}, \texttt{b1})$$

If the modify location set has only one element then the element must be modified, so a ϕ-function does not need to be inserted. This extension to SSA form allows us to treat de-referenced pointers in exactly the same manner as scalars.

3.5.2 Pointer Bitwidths

With a pointer analysis package we can also determine the bitwidth of a pointer. This is useful if we are compiling to a non-conventional device such as an FPGA where memories are segmented into many small chunks [3]. For instance, if we know that a pointer always points to an array of a statically known size, we can set the bitwidth of the pointer accordingly.

4 Bitwise Results

We have implemented the compiler infrastructure, forward propagation, and sequence detection algorithms described in this paper. We are in the process of implementing the

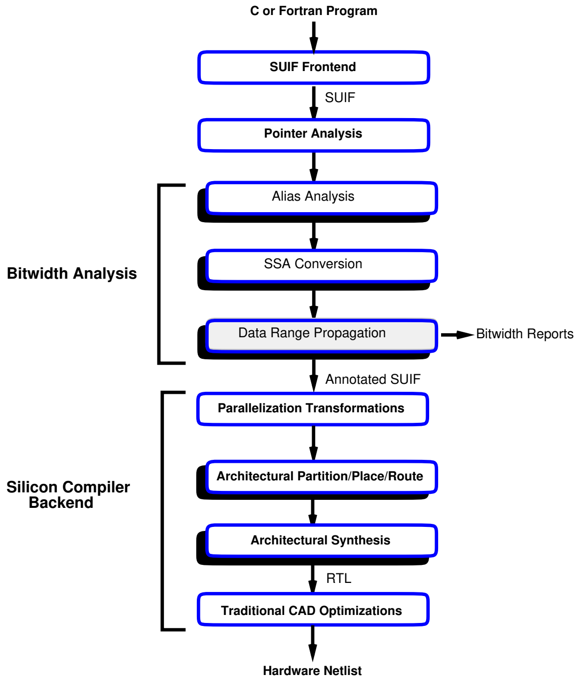

Figure 8: Compiler Flow: Includes general SUIF, Bitwise, Silicon, and CAD processing steps.

back propagation algorithm. In this section we report general results from a standalone *Bitwise Compiler* composed of the first five SUIF passes shown in Figure 8. Further results, after processing with the Silicon Compiler backend, are presented in Section 5 and Section 6.

The frontend of the compiler takes as input a program written in C or FORTRAN and produces a bitwidth-annotated SUIF file. After parsing the input program into SUIF, the compiler performs traditional optimizations and then pointer analysis. We use Radu Rugina's SPAN [15]. Next come the three passes labeled "Bitwidth Analysis". These three passes are the realization of the algorithms discussed in this paper, with the exception of backward propagation. In total, they comprise roughly 12,000 lines of C++ code. We first discuss the bitwidth reports that are generated after these passes, without further backend processing.

To continue, the pointer analysis information is supplied to the following alias information pass, which performs simple interprocedural analysis. From here the SUIF intermediate representation is converted to SSA form. This SSA conversion also implements the SSA extensions of Section 3.5 and Section 3.4. Finally, the data range propagation pass is invoked to produce bitwidth-annotated SUIF along with the appropriate bitwidth reports.

4.1 Experiments

Because the compiler is new, we do not yet compile programs with recursive procedure calls. In the short term, this restriction limits the complexity of the our benchmarks set for general purpose computing. However, it provides adequate support of programs for high-level silicon synthesis.

Table 3.5.2 lists the benchmarks presented in this section. Because multimedia applications are becoming so prevalent, we chose several. We also chose standard applications that exhibit bit and byte-level granularity: softfloat and life.

4.2 Register Bit Elimination

Figure 9 shows the percentage of the original register bits remaining in the program after *Bitwise* has been run. Register bits are used to store static program variables. The lower bound — which was obtained by profiling the code — is included for reference. For the particular data sets supplied to the benchmark, this lower bound represents the fewest possible number of bits needed to retain program correctness, and thus the best any static analysis could possibly achieve. The graph assumes that each variable is assigned to its own register. This is not always the case because a register allocator may lose some of the gains of the analysis by allocating the same register to different sized operands. Nonetheless, this is a useful metric because register bitwidths may still affect functional unit size, data path bitwidths, and switching activity.

Our analysis dramatically reduces the total number of register bits needed. In most cases, the analysis is near optimal, which is especially exciting for applications that perform abundant multi-granular computations. For instance, *Bitwise* almost matches the lower bound for life and mpegcorr, both of which are bit and byte-level applications.

The only application in the figure with substantially suboptimal performance is median. In this case, the analyzer was unable to determine the bitwidth of the input data, thus variables that were dependent on the input data assumed the maximum possible bitwidths ($\perp_{BW}$).

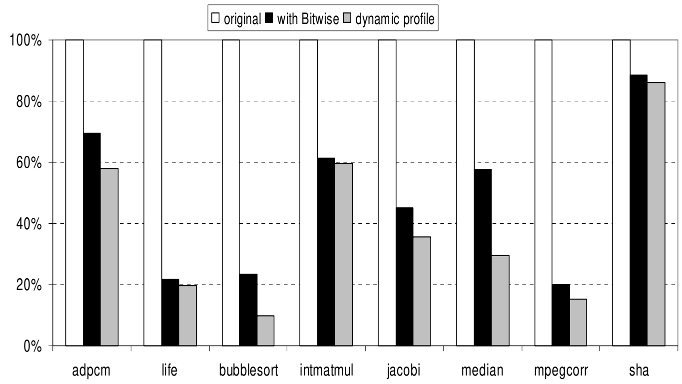

Figure 9: Percentage of total register bits remaining: Original versus post-bitwidth analysis and dynamic profile-based lower bound.

4.3 Memory Bit Elimination

Figure 10 shows the percentage of the original memory bits remaining in the program. Here memory bits are defined as data allocated for static arrays and dynamically allocated variables. This is an especially useful metric when compiling to non-conventional devices such as an FPGA, where memories may be segmented into many small chunks. In addition, because memory systems are one of the primary consumers of power in modern processors, this is a useful metric for estimating power consumptionn [10].

In almost all cases, the analyzer is able to determine near-optimal bitwidths for the memories. There are a couple of contributing factors for *Bitwise*'s success in reducing array bitwidths. First, many multimedia applications initialize static constant tables which represent a large portion of the memory savings shown in the figure. Second, *Bitwise* capitalizes on arrays of Boolean variables.

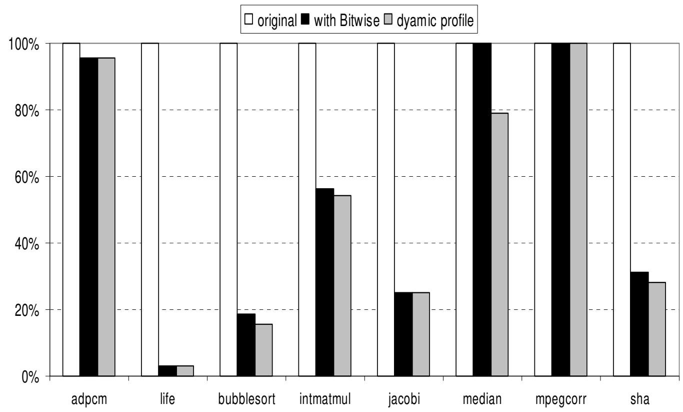

Figure 10: Percentage of total memory remaining: Original versus post-bitwidth analysis and dynamic profile-based lower bound.

4.4 Bitwidth Distribution

It is interesting to categorize variable bitwidths according to grain size. The stacked bar chart in Figure 11 shows the distribution of variable bitwidths both before and after bitwidth analysis. We call this distribution a *Bitspectrum*. To make the graph more coherent, bitwidths are rounded up to the nearest typical machine data-type size. In most cases, the number of 32-bit variables is substantially reduced to 16, 8, and 1-bit values.

For silicon compilation, this figure estimates the overall register bits that can be saved. As we will see in the next sections, reducing register bits will results in smaller datapaths and subsequently smaller, faster, and more efficient circuits. For multimedia applications, the spectrum shows which applications will have the best prospect for packing values into sub-word instructions.

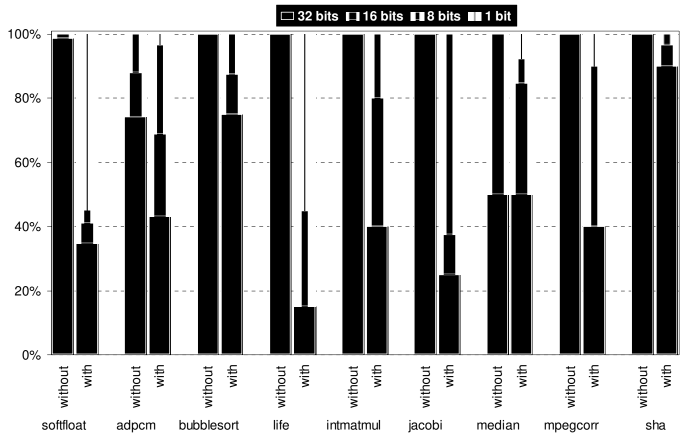

Figure 11: Bitspectrum. This graph is a stacked bar chart that shows the distribution of bitwidths for each benchmark. Without bitwidth analysis, almost all bitwidths are 32-bits. With Bitwise, many widths are reduced to the 16, 8, 1 bit machine types, as denoted by the narrower 16, 8, and 1 bit bars.

5 DeepC Silicon Compiler

So far we have shown that bitwidth analysis is a generally effective optimization and that our *Bitwise Compiler* is capable of performing this task well. We now turn to a concrete application. We have applied bitwidth analysis to the very difficult problem of silicon compilation. For lack of space, we must give the problem of silicon compilation a very brief treatment (in the remainder of this section) and then focus our attention (Section 6) on the impact of bitwidth analysis in this context.

5.1 Overview

We have integrated *Bitwise* with the *DeepC Silicon Compiler* [3], a research compiler under development that is capable of translating sequential applications, written in either C or FORTRAN, directly into a hardware netlist. The compiler automatically generates a specialized parallel architecture for every application. To make this translation feasible, the compilation system incorporates both the latest code optimization and parallelization techniques as well as modern hardware synthesis technology. Figure 8 shows the details of integrating *Bitwise* into *DeepC's* overall compiler flow. Detailed steps of the compiler that are unimportant to our discussion are compressed into a few black boxes. After reading in the program and performing traditional compiler optimizations and pointer analysis, the bitwidth analysis steps are then invoked. These steps were described in detail in Section 4. The silicon compiler backend follows these steps. First, additional parallelization transformations are applied, followed by a high-level architectural partition, place, and route which forms parallel communication threads. Then an architectural synthesis step translates these threads into custom hardware. Following these transformation, traditional computer-aided-design (CAD) optimizations are applied to generate the final hardware netlist. In the flowchart, the raised steps are new *Bitwise* or *DeepC* passes, and the remaining steps are re-used from previous SUIF compiler passes.

5.2 Implementation Details

The *DeepC Compiler* is implemented as a set of over 50 SUIF passes followed by commercial RTL synthesis. The current implementation uses the latest version of Synopsys Design Compiler and FPGA compiler for synthesis. A large set of the SUIF passes are taken directly from MIT's Raw compiler [12], whose backend is in turn built on Harvard's MachSUIF compiler [16]. The backend verilog generator is implemented on top of Stanford's VeriSUIF [6] data structures. Despite the large number of SUIF passes, the majority of the compiler's run-time is consumed by CAD synthesis tools.

5.3 Usage

There has been a limited released of the compiler and it is in use by researchers at MIT and Princeton for reconfigurable computing research, and the University of Massachusetts for system-on-a-chip research. When used for reconfigurable computing, the compiler is coupled with further silicon compilation tools, such as the VirtuaLogic [9] emulation system from IKOS, or software and drivers for Annapolis System's WildCard PCMCIA card. For use in custom chip design, downstream tools must be capable of accepting a logic netlist and placing and routing that design into a specific silicon substrate.

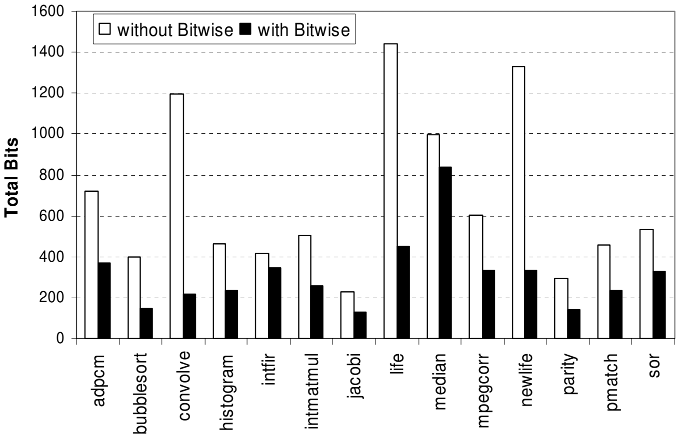

Figure 12: Register Bits After Bitwise Optimization. In every case *Bitwise* saves substantial register resources in the final silicon implementation.

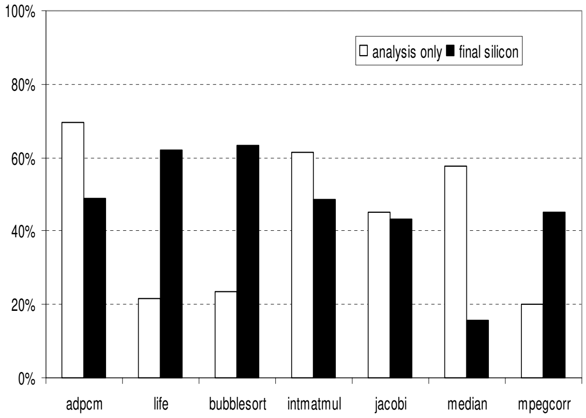

Figure 13: Register Bit Reduction, After High Level Analysis versus Final Silicon. The fluctuations in bitwidths savings between final silicon and high-level analysis is do to factors such as variable renaming and register allocation.

6 Impact on Silicon Compilation

In this section we characterize the impact of bitwidth analysis on silicon compilation. As described in the previous section, the *DeepC Silicon Compiler* has the opportunity to specialize memory, register, and datapath widths to match application characteristics — we expect bitwidth analysis to have a large impact in this domain. However, because backend CAD tools already implicitly perform some bitwidth calculation during optimizations such as dead logic elimination, accurate measurements require end-to-end compilation. We need to compare final silicon both with and without *Bitwise*.

We introduce our benchmarks in the next section, and then describe the dramatic area, latency, and power savings that bitwidth analysis enables[2].

6.1 Experiments

We present experimental results for an initial set of applications that we have compiled to hardware. For each application, our compilation system produces an architecture description in RTL Verilog. We further synthesize this architecture to logic gates with a commercial CAD tool (Synopsys). In this paper we report area and speed results for Xilinx 4000 series FPGAs, and power results for IBM's SA27E process – a 0.12 micron, 6-layer copper, standard-cell process.

The benchmarks used for silicon compilation are included in Table 3.5.2. These applications are mostly short benchmarks, but include many multimedia kernels. It is important to note that the relative small size of the benchmarks is dictated by the current synthesis time of our compilation approach and not *Bitwise*.

6.2 Registers Saved in Final Silicon

We first compiled each benchmark into a netlist capable of being accepted by either Xilinx or IBM CAD tools to produce "final silicon". The memory savings reported in Section 4 translate directly into silicon memory savings when we allow a separate small memory for each program variable. This small memory partitioning process is further described in earlier work [3].

[2]Note that we also found considerable synthesis compile time savings which are not reported here.

Register savings, on the other hand, vary as additional compiler and CAD optimizations transform the program's variables. Variable renaming and register allocation also distort the final result by placing some scalars in more than one register and others in a shared register. Figure 12 shows the total FPGA bits saved by bitwidth optimization. For Xilinx FPGA compilation, the fixed allocation of registers to combinational logic will distort the exact translation of this savings to chip area, as some registers may go unused.

Our findings are very positive — the earlier bitwidth savings translated into dramatic savings in final silicon, despite the possibilities for loss of this information or potential overlap with other optimizations. However, because there is not a one-to-one mapping from program scalars to hardware registers, the exact savings do not match. Examining Figure 13, we see that the percentage of bits saved by high-level analysis are sometimes greater and sometimes less than those bits saved in final silicon. We explain these differences as follows. First, there are many compiler and CAD passes between high-level analysis and final silicon generation. If in any of these passes the bitwidth information is "lost", for example when a new variable is cloned, then the full complement of saved bits will not be realized. On the other hand, the backend passes, especially the CAD tools, are also attempting to save bits through logic optimizations. Thus these passes may find saving that the current high-level pass is not finding. Finally, variable renaming and register sharing also change the percentages.

6.3 Area

Registers saved translate directly into area saved. Area savings also result from the reduction of associated datapaths. Figure 14 shows the total area savings with Bitwise optimizations versus without. We save from 15% to 86% in overall silicon area, nearly an 8× savings in the best case.

Note that in the DeepC Compilation system pointers do not require the full complement of 32-bits. Using the MAPS [4] compiler developed for Raw, arrays have been assigned to a set of *equivalence classes*. By definition, a given pointer can only point to one equivalence class, and thus needs to be no wider than $\log \sum_a S_a$, where S_a is the size of each memory array specified in the equivalence class. This technique is further described in [2].

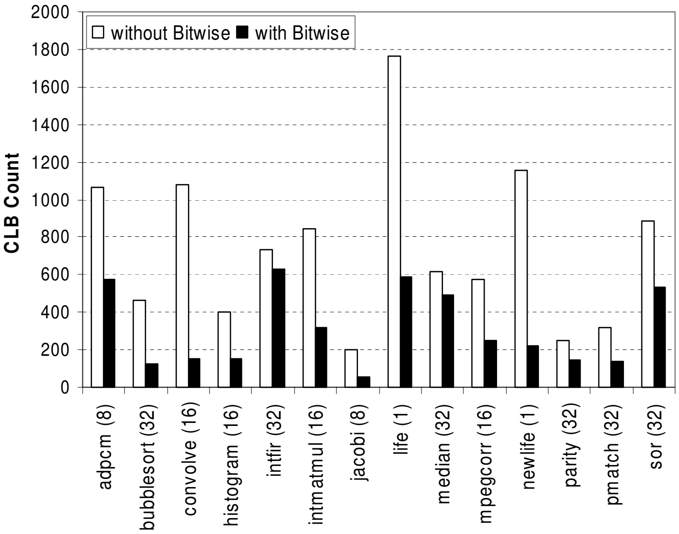

Figure 14: FPGA Area After Bitwise Optimization. Register savings translate directly into area savings for FPGAs. In the figure, CLB count measures the number of internal combinational logic blocks required to implement the benchmark when compiled to FPGAs. Combinational logic blocks (CLBs) each include 2 four input lookup tables and 2 flip-flop registers. The number in parenthesis by each benchmark is the resulting bitwidth of the main datapath.

6.4 Clock Speed

We also expect bitwidth optimization to reduce the latency along the critical paths of the circuit and increase maximum system clock speed. If circuit structures are linear, such as a ripple carry adder, then we would expect a linear increase. However, common structures such as carry-save adders, multiplexors, and barrel shifters are typically implemented with logarithmic latency, thus bitwidth reduction translates into a less-than-linear but significant speedup. Figure 15 shows the results for a few of our benchmarks. The largest speedup is for convolve, in which the reduction of constant multiplications increased clock speeds by nearly 3×. On the other hand, the MPEG correlation kernel did not speed up because the original bitwidths were already close to optimal.

6.4.1 Power

As expected, the area saved by bitwidth reduction translated fairly directly into power savings. Our first hypothesis was that these saving might be lessened by the fact that inactive registers and datapaths would not consume power. We were incorrect. The muxes and control logic leading to these registers still consume power. Figure 16 shows the reduction in power achieved. In order to make these power measurements, we first ran a verilog simulation of the design to gather switching activity. This switching activity records when each register toggles in the design. This information is then used by logic synthesis, along with an internal zero delay simulation, to determine how often each wire changes state. The synthesizer then reports average dynamic power consumption, in milliWatts, which we report here. In the current implementation we do not use bitwidth analysis to reduce the total cycle count, and thus total energy is reduced proportionately.

We measured power for bubblesort, histogram, jacobi, pmatch, and newlife. Newlife had the largest power savings,

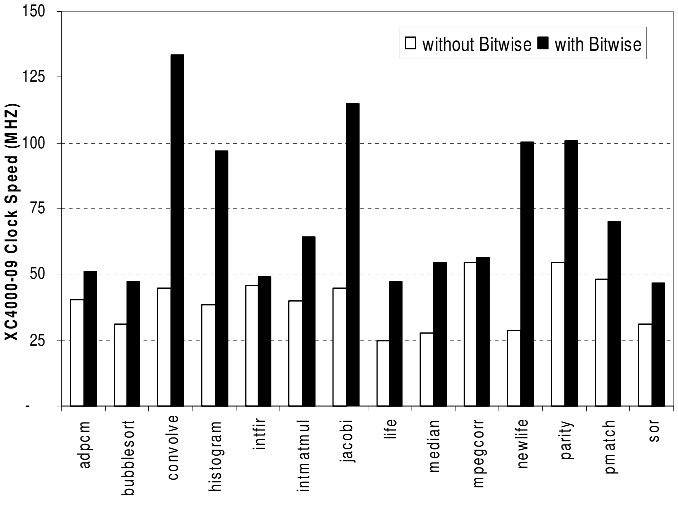

Figure 15: FPGA Clockspeed After Bitwise Optimization. Benchmarks are universally faster after bitwidth analysis when compiled to Xilinx XC4000 FPGAs (-09 speed grade) with Synopsys. The actual number of CLBs on the critical paths, ranging from 15-38 before bitwidth optimization and 7-16 afterwards, is the key factor in determining clock speed.

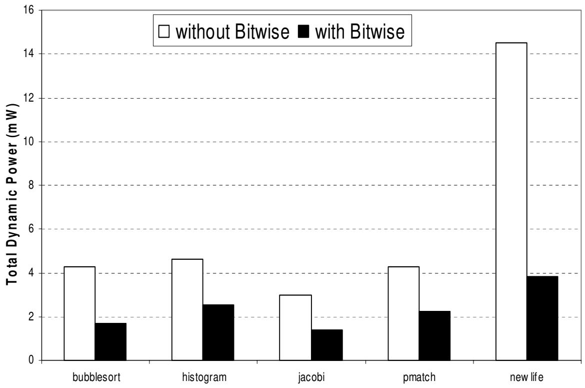

Figure 16: ASIC Power After Bitwise Optimization. Here we assume a 200MHZ clock for the .12 micron IBM SA27E process. The total cycle count (number of clocks ticks to complete each benchmark) is not affected by bitwidth, and thus total energy will scale proportionally.

reduced from 14 mW to 4 mW, while the other four benchmarks had more conservative (but still very high!) power savings. We expect that at least a portion of these savings can be translated to the processor regime — in which power consumption is typically hundreds of times higher — if future low power architectures can successfully take advantage of a-priori bitwidth information.

6.5 Discussion

For reconfigurable computing applications, savings can easily be a "make or break" difference when comparing performance per area to that of traditional processors. Because FPGAs provide an additional layer of abstraction (emulated logic), it is important to compile-through as many higher levels of abstraction as possible. Statically taking advantage of bitwidth optimization is a form of partial evaluation which can help to make FPGAs competitive with more traditional, but less adaptive, computing solutions. Bitwidth analysis is a key technology *enabler* for FPGA computing.

For ASIC implementations, bitwidth savings will directly translate into reduced silicon costs. Of course, many of these cost savings could be captured by manually specifying more precise variable widths — but manual optimization comes at the cost of manual labor. Additionally, reducing the probability of errors is invaluable in an ASIC environment, where companies who miss with first silicon often miss entire market windows. As we approach the billion transistor era, raising the level of abstraction for ASIC designers will be a requirement, not a luxury.

7 Related Work

Brooks et.al., dynamically recognize operands with narrow bitwidths to exploit sub-word parallelism [5]. Their research confirms our claim that a wide range of applications— particularly multimedia applications— exhibit narrow bitwidth computations. Using their techniques, they are able to detect and expoit bitwidth information that is not statically known. However, because they are detecting bitwidths dynamically, their research cannot be applied to applications that require a priori bitwidth information.

Scott Ananian also recognized the importance of static bitwidth information [1]. He uses bitwidth analysis in the context of a Java to silicon compiler. Because bitwidth analysis is not the main thrust of his research, he uses a simple data flow technique that propagates bitwidth information. Our method of propagating data-ranges is a more precise method for discovering bitwidths.

The data-range propagation techniques presented by Jason Patterson [14] and William Harrison [8] are similar to those presented in this paper. While their work proved to be effective, they did not consider backward propagation which offers abundant data-range information. Furthermore, their techniques for discovering loop-carried sequences do not include the general methods discussed in this paper.

8 Conclusion

In the paper we have formalized bitwidth analysis as a value range propagation problem. With a new suite of bitwidth extraction techniques, we demonstrate bi-directional bitwidth propagation as well as a closed form solution for finding bitwidth information in the presence of loops. Our initial results are promising — compile-time analysis approaches the accuracy of run-time profile-based analysis. When incorporated into a silicon compiler, bitwidth analysis dramatically reduces the logic area by 15%-86%, improves the clock speed by 3%-249%, and reduces the power by 46%-73% of the resulting circuits. We anticipate many future uses of this technique, including compilation for SIMD architectures and compilation for low power.

References

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

[2] J. Babb. *High-Level Compilation For Reconfigurable Architectures.* PhD thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, February 2000.

[3] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications Into Silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa Valley, CA*, April 1999.

[4] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[5] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performanc. January 1999.

[6] R. French, M. Lam, J. Levitt, and K. Olukotun. A General Method for Compiling Event-Driven Simulations. *32nd ACM/IEEE Design Automation Conference*, June 1995.

[7] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. Technical report, Oregon Graduate Institute of Science and Technology, 1995.

[8] W. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, 3:243–250, May 1977.

[9] IKOS Systems, Inc. *VirtuaLogic Emulation System Documentation*, 1999. Version 3.0.4.

[10] J. Kin, M. Gupta, and W. H. Magione-Smith. The filter cache: An energy efficient memory structure.

[11] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. Technical report, Digital Cambridge Research Laboratory, 1998.

[12] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, Oct. 1998.

[13] Open SystemC Initiative. *systemc.org.*

[14] J. Patterson. Accurate Static Branch Prediction by Value Range Propagation. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 37(11):67–78, June 1995.

[15] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, pages 77–90, Atlanta, GA, May 1999.

[16] M. D. Smith. Extending SUIF for Machine-dependent Optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, Jan. 1996.

[17] 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. *ACM SIGPLAN Notices*, 29(12), Dec. 1996.