Compile-Time Area Estimation for LUT-Based FPGAs

DHANANJAY KULKARNI and WALID A. NAJJAR
University of California, Riverside
ROBERT RINKER
University of Idaho
and
FADI J. KURDAHI
University of California, Irvine

The Cameron Project has developed a system for compiling codes written in a high-level language called SA-C, to FPGA-based reconfigurable computing systems. In order to exploit the parallelism available on the FPGAs, the SA-C compiler performs a large number of optimizations such as full loop unrolling, loop fusion and strip-mining. However, since the area on an FPGA is limited, the compiler needs to know the effect of compiler optimizations on the FPGA area; this information is typically not available until after the synthesis and place and route stage, which can take hours. In this article, we present a compile-time area estimation technique to guide SA-C compiler optimizations. We demonstrate our technique for a variety of benchmarks written in SA-C. Experimental results show that our technique predicts the area required for a design to within 2.5% of actual for small image processing operators and to within 5.0% for larger benchmarks. The estimation time is in the order of milliseconds, compared with minutes for the synthesis tool.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids; B.7.1 [**Integrated Circuits**]: Types and Design Styles; B.7.2 [**Integrated Circuits**]: Design Aids; C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms: Measurement, Performance

Additional Key Words and Phrases: Reconfigurable computing, resource estimation, compiler optimization

This research was supported by National Science Foundation (NSF) ITR Award 0083080 and by DARPA under US Air Force Research Laboratory contracts F33615-98-C-1319.
Authors' addresses: D. Kulkarni and W. A. Najjar, University of California, Riverside, Department of Computer Science and Engineering, Riverside, CA 92521; email: {kulkarni,najjar}@cs.ucr.edu; R. Rinker, University of Idaho, Computer Science Department, Moscow, ID 83844; email: rinker@cs.uidaho.edu; F. J. Kurdahi, University of California, Irvine, Department of Electrical and Computer Engineering, Irvine, CA 92717; email: kurdahi@ece.uci.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1084-4309/06/0100-0104 $5.00

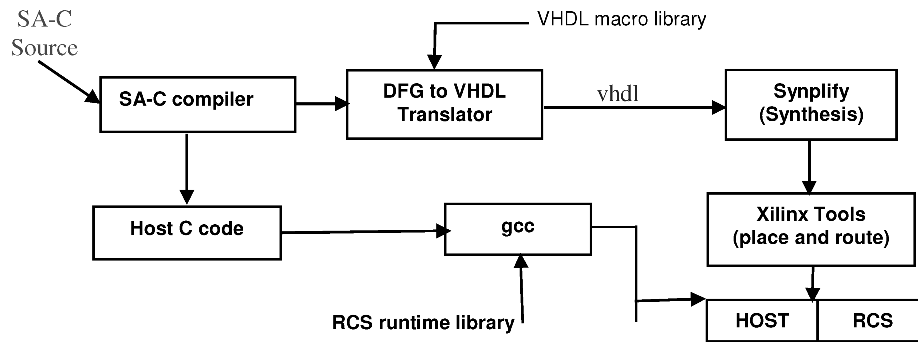

Fig. 1. Cameron project.

1. INTRODUCTION

Even though FPGAs present a computational density advantage over general-purpose processors [DeHon 2000] for certain applications, the biggest obstacle to the widespread use of FPGA-based reconfigurable computing lies in the difficulty of programming them. A typical design cycle for programming FPGAs starts with a behavioral or structural description of the design, using hardware description languages (HDLs) such as VHDL[1] or Verilog. These HDLs require the programmer to explicitly handle the issues of timing and synchronization of the complete design. Most application program developers are more familiar with an algorithmic programming paradigm and are not experts in HDL programming. The goal of the Cameron Project [Najjar et al. 2003; Rinker et al. 2001; Böhm et al. 2002b] is to bridge the semantic gap between applications and FPGAs by providing an algorithmic language called SA-C (Single Assignment C, pronounced *sassy*) that is suitable for mapping image processing applications [Böhm et al. 2002a; Hammes et al. 2001b] onto reconfigurable systems. The ease of programming in SA-C makes FPGAs and other adaptive computer systems more readily available to application programmers.

An overview of the SA-C compilation process is shown in Figure 1. The compiler translates source code into dataflow graphs (DFGs), which can be viewed as abstract hardware circuits without timing information. The SA-C DFG-to-VHDL translator converts the DFG into VHDL, which is processed using commercial synthesis tools to produce an FPGA configuration. The SA-C compiler also generates the necessary run-time host code to direct the execution of the program on the reconfigurable processor.

The SA-C compiler applies extensive optimizations [Hammes et al. 2001a; Draper et al. 2001; Böhm et al. 2001] to the code before producing the DFG. Like any optimizing compiler, these optimizations often produce a design that is structurally different than what is implied in the source program; this may have a significant impact on FPGA resource usage. Some optimizations produce better (faster and/or smaller) designs in all cases, while others trade off space vs. time, and can be selectively activated by the programmer. Unfortunately, since the SA-C compiler produces dataflow graphs that must be processed by

[1]VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.

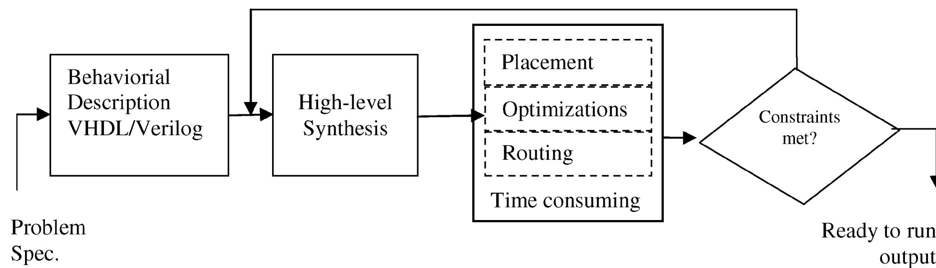

Fig. 2. Typical design flow targeted for FPGAs.

synthesis tools, time and space information about a particular design is not available until minutes or hours after compilation. In this article, we describe a *compile-time* estimation approach applied during the compilation process. Our technique makes resource estimations available before the DFG-to-VHDL translation, and takes several orders of magnitude less time to compute than the commercial synthesis tools. This speed-up allows the estimation to serve as feedback to aid in further optimizations. Experimental results show that our technique achieves estimates within 2.5% of the actual design for small image-processing operators and 5.0% for larger benchmarks. An earlier version of this work was published in Kulkarni et al. [2002].

The rest of the article is organized as follows: the next section discusses the motivation for the estimation, while Section 3 presents the details of the SA-C compiler and the dataflow graphs. Compile-time estimation approach is presented in Section 4. Experimental results are presented in Section 5. References to related work are given in Section 6, and Section 7 concludes and describes some future work.

2. MOTIVATION

Figure 2 shows a traditional design flow used for FPGAs. First, a behavioral description of the design is coded in a hardware definition language (HDL) such as VHDL or Verilog. Next, the code is processed by high-level synthesis tools, followed by placement and routing. If the resulting design does not meet necessary space and timing constraints, it is reworked and the entire process is repeated. The SA-C language allows the problem to be expressed in an algorithmic language, which is a method more familiar to most application programmers. This considerably shortens the “top end” of the design cycle. However, the time-consuming synthesis, placement and routing phases remain in the design loop—the designer must patiently wait for these steps to finish before knowing the results of the chosen algorithm and optimizations. This motivates us to look at a compile-time area estimation approach in the SA-C compilation process.

3. SA-C COMPILER AND DATAFLOW GRAPHS

3.1 Unique Features of SA-C

SA-C is an expression-oriented, single-assignment (functional) language whose design facilitates translation into hardware descriptions. The compiler can

```
uint8[:,:] main (uint8 Arr[:,:]) {
        uint8 r[:,:] =
                for window W[3,3] in Arr {
                        uint8 s = array_sum(W);
                        uint8 v = if(s>100) return (s-100)
                        else return (s);
                } return(array(v));
} return( r )
```

Fig. 3. Example of a SA-C program.

readily analyze the source code and extract both fine-grained and coarse-grained parallelism. SA-C's syntax is roughly based on C; however, there are significant differences as well, mostly due to its use as a hardware generation language. Unique features of the SA-C language include:

—A flexible *type* system, including signed and unsigned integers of arbitrary bit-width, and fixed-point numbers.
—Multi-dimensional arrays whose type includes the array size and shape. The compiler can use this type information about the array to optimize operations on the array.
—No pointers and no recursion, designed to prevent programmers from applying Von Neumann models[2] that do not map well onto FPGAs.
—Loop *generators*, which are used in place of the more traditional "loop index used as an array subscript" to perform operations on arrays.
—Reduction operators that can be applied to data produced in loop bodies, such as array sum or histogram. Efficient VHDL implementations can be used for these operators.

A simple SA-C program is shown in Figure 3. This somewhat simplistic program accepts a 2-D array (named *Arr*) of 8-bit unsigned integers (i.e., type *uint8*) as input. A window generator (*for window*$\cdots$) extracts all 3×3 subarrays from the array and sums the elements in each subarray. A new array (r) is created, with each element being either the sum of the corresponding window or, if the sum is greater than 100, the sum minus 100.

3.2 SA-C Compiler Optimizations

Figure 4 shows the SA-C compilation process, including the estimation tool. The user can direct the compiler to perform numerous optimizations before generating the DFG. Several traditional optimizations—including code motion, constant folding, array and constant value propagation and common subexpression elimination—minimize the calculations required by the hardware, and always result in better (faster and/or smaller) code. Function inlining and loop unrolling increase parallelism—they *usually* cause an improvement in the code. Another set of optimizations (e.g., strip-mining and loop fusion) trade-off speed

[2]Von Neumann model is the traditional computer architecture model, which consists of the input unit, output unit, ALU, control unit and the memory unit.

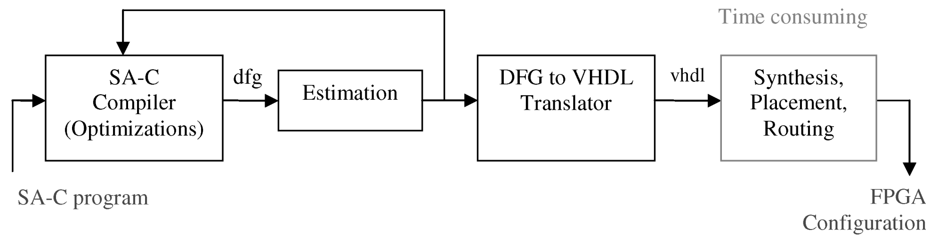

Fig. 4. Compile-time estimation.

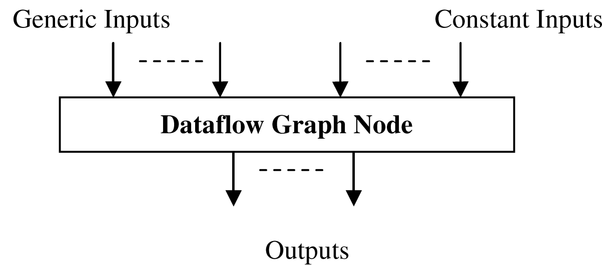

Fig. 5. Generic dataflow graph node.

at the cost of area. The optimizations interact with each other—some optimizations don't improve the code themselves, but they enable other optimizations. Sometimes a seemingly minor optimization can cause significant changes in the final design. The resource estimation tool allows the user to quickly see the effects of the selected optimizations.

3.3 Dataflow Graphs and Execution Model

Dataflow graphs can be viewed as abstract hardware circuit diagrams without timing considerations. The functional elements of a DFG are nodes, shown in Figure 5. Each node is characterized by a node type, one or more inputs, and one or more outputs. A DFG suggests a well-defined execution semantic—the node executes ("fires") when all of its input values (called "tokens") arrive; the firing consumes one value from each input. The output depends only on these inputs. This execution model works well to describe hardware behavior.

Most DFG nodes represent combinational logic, including arithmetic and bitwise logical operations, selection (to choose a result from several possibilities), and I/O. However, another group, the generator and reduction nodes, which implement loop operations, are implemented as sequential processes. These operations require multiple clock cycles, one or more state machines and coordination between nodes.

Traditional processors provide a fairly small instruction set that is used to write applications, and can be executed on the hardware. In contrast, FPGAs consist of an amorphous mass of configurable cells that can be interconnected in a large number of ways. In order to limit this number, an *abstract machine* is defined. The example shown in Figure 6 serves as a reasonable target for the compiler during the translation process. The compiler generates some of the

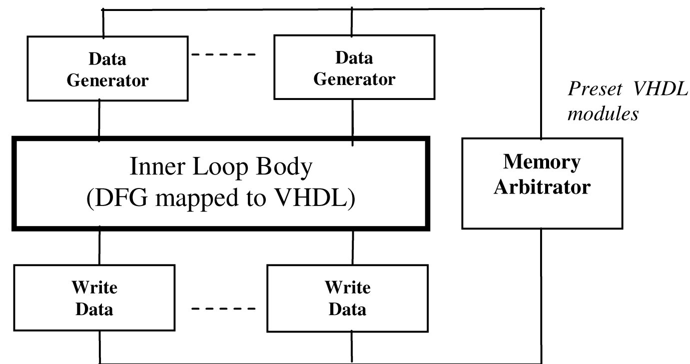

Fig. 6. Abstract state machine mapped onto FPGA.

modules at compile-time, while others are pre-built modules and are included from a library in the final design.

The DFG completely specifies the *inner loop body* (ILB), which is purely combinational and is built from scratch for each program. A DFG-to-VHDL translator converts the DFG into VHDL, and also selects the necessary additional VHDL modules from a run-time library to complete the system; these modules are parameterized by values determined from the DFG. The *memory arbitrator* does the initialization and scheduling of memory accesses, and handles resource contention. The *data generator*, *write-data* and *memory arbitrator* modules are sequential processes that establish the synchronization and timing within the system. The translator also creates a "glue" module that wires the other modules together into a complete system.

The resulting abstract machine executes roughly as follows: data generator(s) sequentially supply values to the ILB. After waiting for signals to propagate through the ILB, the result values are written to memory. The data generator and write-data modules are synchronized so that neither section gets ahead of the other. Usually the data generators are retrieving data from one area of memory while the write-data modules are writing to another; the memory arbitrator imposes the most efficient ordering of these overlapping memory operations.

4. ESTIMATION APPROACH

4.1 Compile-Time Estimation

The abstract machine described above serves as the basis for our resource estimation, since it is created in the final step of the compilation process, just before synthesis and routing occurs. It is relatively easy to precompute the resource usage of the prebuilt modules as a function of the parameters and to save the results in a table. On the other hand, the ILB is unique to each SA-C program. For these nodes, we apply an estimation technique based on general formulas. Since the compiler optimizations mainly affect the structure of the ILB, we estimate the effect of such optimizations on the size of the ILB.

As shown in Figure 4, we apply the estimation model to the DFG and feed the estimation results back to the compiler to aid in further optimizations. Our focus is to provide quick and reasonably accurate resource usage estimation of a SA-C program, so the compiler user can use the estimation feedback to develop a SA-C program that produces an efficient design that fits on the FPGA.

4.2 Estimation Methodology

Since the DFG does not contain any low-level structural or timing information, our estimation tool does not incorporate scheduling, resource allocation or binding algorithms—the estimation considers only the impact of logic synthesis tools that are applied to the DFG description. Our estimation method uses a set of general formulae for each type of DFG node at compile-time to estimate the resource usage of the SA-C program. The general formula and its coefficients are stored in a data file called *nodeparams*. Input to the estimator is the DFG and this file; the output is the estimated resource usage of the dataflow graph. The complexity of the estimation algorithm is $O(\mathrm{n})$, where n is the number of nodes in the DFG.

The following procedure was used in determining the parameter values used by the estimation model:

(1) Vary the parameters of each type of DFG node and create the corresponding VHDL instances.
(2) Synthesize these VHDL instances and record the estimations reported by Synplify.[3]
(3) Do a regression analysis on the estimation values reported in step 2 to obtain the coefficients for the node. We do simple linear and nonlinear regression using the NLREG package.
(4) Record the coefficients in the *nodeparams* file.

Synthesis tools can be directed to optimize a design in a particular way—a trade-off of speed vs. space, for example. The estimator currently only uses the "default" implementation. Multiple implementations could be supported by storing separate parameters for each implementation.

During compile-time, the estimation program uses the coefficients recorded for each node to calculate an estimate of the area used by the DFG. Clearly, this method does not capture the exact timing as produced by the place-and-route. One question to answer is whether this approach will produce estimates that are "close enough"—that is, estimates that allow different optimizations to be considered quickly, yet are accurate enough to provide meaningful feedback.

4.3 Approximation Formulas

The general formulas are parameterized as follows: The value y is the estimated LUT usage and is typically a function of bit-width (x) and number of input values (z). Parameters C, p0, p1, p2, c0, c1 are coefficients that result from the regression analysis; they are recorded in the *nodeparams* file.

[3]Synplify is a widely used logic synthesis tool developed by Synplicity Inc.

Table I. Nodes using Linear Approximation

Name	No. of Inputs	Description
UADD/IADD	2	Unsigned/signed addition of I_0 and I_1
USUB/ISUB	2	Unsigned/signed subtraction of I_0 and I_1
ULT/ILT	2	Unsigned/signed < comparison of I_0 and I_1
ULE/ILE	2	Unsigned/signed <= comparison of I_0 and I_1
UGT/IGT	2	Unsigned/signed > comparison of I_0 and I_1
UGE/IGE	2	Unsigned/signed >= comparison of I_0 and I_1
UEQ/IEQ	2	Unsigned/signed equality comparison of I_0 and I_1
UNE/INE	2	Unsigned/signed inequality comparison of I_0 and I_1
BIT-AND	2	Bit-wise AND of I_0 and I_1
BIT-OR	2	Bit-wise OR of I_0 and I_1
BIT-EOR	2	Bit-wise exclusive OR of I_0 and I_1

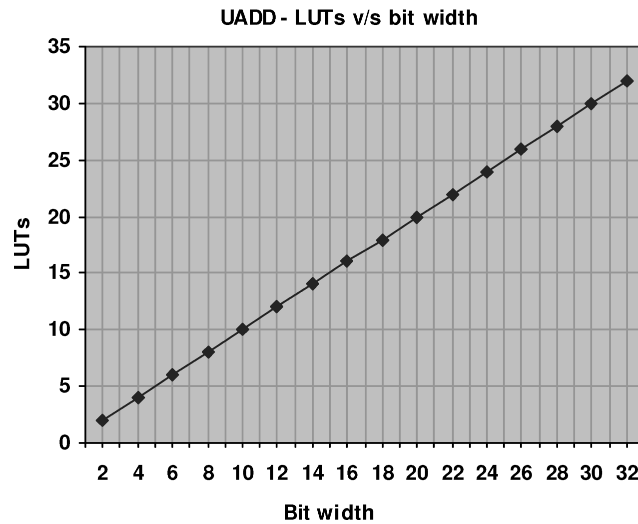

Fig. 7. Linear approximation y = p0 + p1*x, where p0 = 0 and p1 = 1.

The space required for the DFG nodes that make up the inner loop body can be approximated by one of the following formulas:

Constant: $y = C$

These nodes synthesize as VHDL signals and provide an interface with the outside world or memory. They consume a fixed amount of resource on the FPGA. Example nodes include:

INPUT, (1 input, 1 output): Gets a value from the input channel specified by input and delivers it to output.

OUTPUT, (2 input, 0 output): Takes a value specified by the first input and connects it to an output channel specified by the second input.

Linear: $y = p0 + p1 * x$

Most arithmetic nodes belong to this category. The resource usage is a linear function of the bit-width (x). DFG nodes shown in Table I perform arithmetic and bit-manipulation operations. The arithmetic nodes implicitly treat their inputs as either signed or unsigned integers. Figure 7 shows

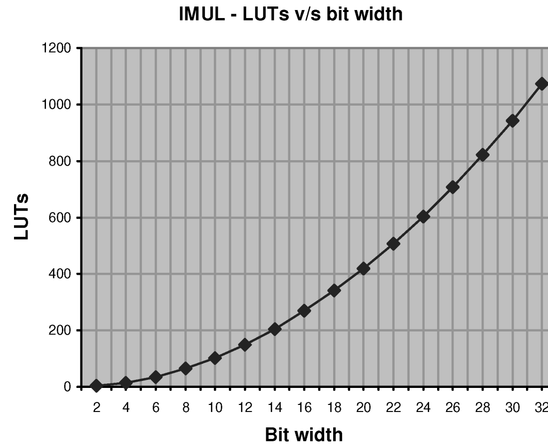

Fig. 8. Quadratic approximation $y = p0 + p1^* (x - p2)^2$, where $p0 = -2.509$, $p1 = .041$, $p2 = -0.125$.

Table II. Nodes using Bi-Product Approximation

Name	No. of Inputs (nvals)	Description
ISUM-MANY	2	Sum the unsigned input values; each input pair represents a value and a Boolean mask; the arithmetic is performed at the output port's bit width
USUM-MANY	2	Sum the signed input values; each input pair represents a value and a Boolean mask; arithmetic is done at the output port's bit width

the LUT usage of UADD node; the other nodes in this group have similar characteristics.

Quadratic: $y = p0 + p1 * (x - p2)^2$

The resource usage of the nodes in this category is a quadratic function of the form $y = \mathbf{p0} + \mathbf{p1}^*(\boldsymbol{x} - \mathbf{p2})^2$, where x is the bit-width of the node. Examples include the multiplication nodes UMUL (unsigned) and IMUL. Figure 8 shows the LUT usage for a 32 bit IMUL node.

Bi-Product: $y = (z\text{-}p0) * (x - p1) + p2$

The resource usage of these nodes depend on the bit-width (x) and the number of input values (z). An example is ISUM-MANY, used to compute the sum of a set of values. Such a node might be employed to implement a loop that computes the sum of a group of array elements after it has been unrolled. Table II lists the multi-input arithmetic nodes that use bi-product approximation. Figure 9 shows the family of curves plotted for bi-product approximation for ISUM-MANY. We speculate that the number of LUTs increases with the bit width because of the increased routing congestion that occurs as a result of the convergence of more signal lines into the LUTs.

MultiLinear2: $y = c0 + (c1 * x/2)(z/2 - 1)$

The resource usage of these nodes depends on bit-width (x) and the number of inputs (z). Table III lists the multi-input logic operator nodes that use

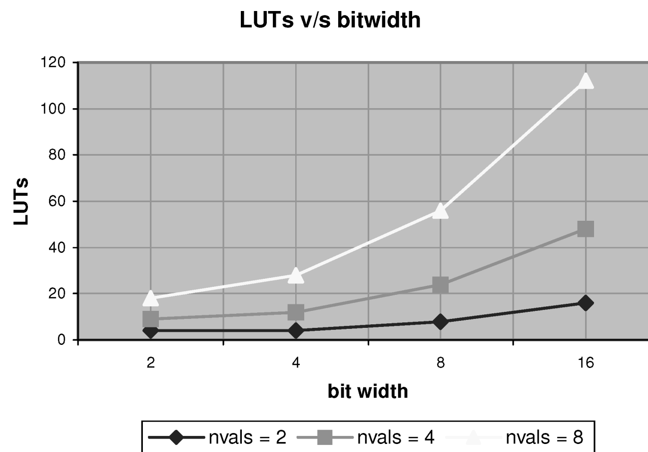

Fig. 9. Curves for Bi-product approximation (ISUM-MANY). *Note*: nvals refers to the number of inputs.

Table III. Nodes using Multilinear2 Approximation

Name	Inputs	Description
AND-MANY	Var	'and' the input values; each input pair represents a value and a Boolean mask
OR-MANY	Var	'or' the input values; each input pair represents a value and a Boolean mask

multilinear2 approximation. These nodes allow an arbitrary number of input values, each with an associated Boolean mask value.

Figure 10 shows the family of function values for an instance of AND-MANY nodes.

4.4 Estimation Heuristics

Most synthesis tools do timing and resource optimization, wherein they try to make the design as small and as fast as possible. These tasks are very complex because they entail the solution of intractable problems. Our approach does not attempt an exact solution to such problems. Rather, our emphasis is to provide quick but reasonably accurate estimation of the resource area, and leave the complexity of the mapping to the synthesis phase. However, an estimator that does not account for some of the optimizations will produce results that are not accurate enough to be useful.

To account for such situations, we introduce heuristics in our compile-time estimation algorithm. These heuristics are based on structural patterns that are frequently produced during synthesis optimization. When such a pattern is found, we replace the estimates for the individual nodes with a single estimate for the pattern. This process requires one additional pass over the DFG and does not increase the complexity of the algorithm. We describe two of these patterns in the following paragraphs.

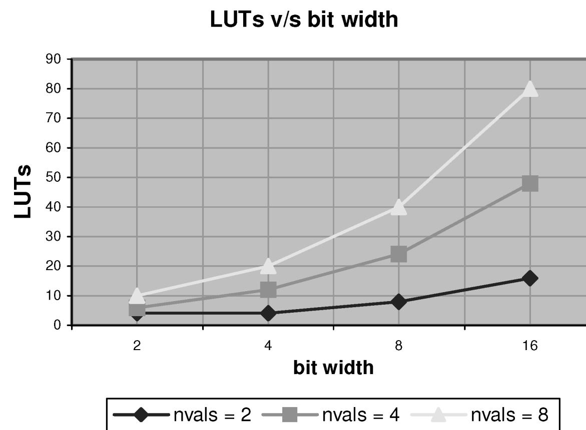

Fig. 10. Curves for multilinear2 approximation (AND-MANY). *Note*: nvals refers to the number of inputs.

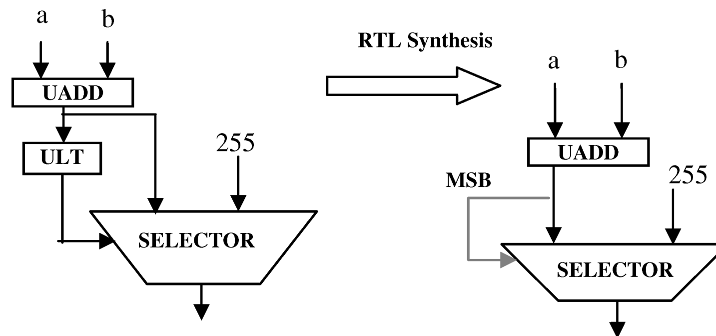

Fig. 11. Comparison nodes optimized during synthesis.

Multiplication by Constants

When a multiplication operation appears in a SA-C program, the compiler generates a UMUL node in the corresponding DFG. Since multiplication is an expensive operation, many but not all are removed via the strength reduction optimization. If the multiplication is by a constant that is a combination of powers of 2, then the synthesis tool optimizes it as a shift operation, thereby eliminating the very expensive gate level description for the UMUL node. To account for this optimization, we identify the pattern at the DFG itself and incorporate the optimization by avoiding the estimation of UMUL node.

Comparison Nodes

In many cases, a node such as ULT (unsigned less than) or UGT (unsigned greater than) is created to perform a general comparison between two values. Depending on the values being compared, the comparison logic can be optimized by the synthesizer. Figure 11 shows the translation of the SA-C statements *val1* $= a + b$*; if (val1 > 255) val2 = 255 else val2 = val1*—this is a common operation in image processing called saturation arithmetic. The output of an unsigned adder (UADD) is compared to check if it is less than or equal to 255

(11111111_b) and this result is used to select the right-hand value in the SELECTOR node. In the optimized circuit, the most significant bit of the adder output is directly fed to the multiplexer.

5. EXPERIMENTAL RESULTS

Three sets of experiments demonstrate the effectiveness of our estimation method. The first set consists of several simple but common image processing (IP) operations. The second set shows application to larger image processing benchmarks. In the third, we explore the impact of discretionary and aggressive compiler optimizations on our estimation of resource usage. All benchmarks used in this paper are compiled for Annapolis Micro System Inc.'s WildStar board [Annapolis 2000]. This board uses the Xilinx Virtex (XCV1000) FPGAs [Xilinx 2000], each with an equivalent of 1 million system gates per chip. There are 27648 4-input LUTs in each FPGA. The results show that our compile-time estimation technique is fast and produces estimates that are reasonably accurate.

5.1 Image-Processing Operators

A set of IP operations were defined during SA-C's development [Draper et al. 2001; Böhm et al. 2001] to evaluate language expressiveness and performance; they were chosen to be representative of simple IP operations that can be used to form more complex applications. For this experiment, we wrote the primitives in SA-C and compiled them as individual programs. Each operator was then synthesized using Synplify. Table IV compares the results with our estimator.

The average execution time for the estimator is only 1 millisecond. Runtime for Synplify ranged from 1 to 5 minutes, with the average being around 2 minutes, for the listed operators. The estimator produces results that are within 2.5% of actual, on average; worst case resulted in an error of just over 6%. We believe these results are accurate enough to serve as a good estimate for the IP operators. Weighted error is calculated as the absolute error between the sum of all the Synplify estimations and the sum of all our compile-time estimations.

5.2 Image-Processing Benchmarks

More complex IP applications are written using the simple image processing operators. Two or more primitives in a sequence provides a greater opportunity for the SA-C compiler to perform optimizations. Optimizations cause the inner loop bodies of the individual routines to be transformed and restructured; thus the resource estimation is a more challenging problem. The following IP applications were used in the experiment:

- *Open* is defined as two steps: *dilation* followed by *erosion. Close* is the reverse process. Since dilation and erosion each involve a loop, the two used in sequence provide an opportunity for loop fusion.
- *Wavelet* is a common image compression algorithm. The version implemented in SA-C is based on the Cohen-Daubechies-Feauveau wavelet algorithm [Cohen et al. 1992].

Table IV. Results for IP Operators

Benchmark	Synplify Estimation (# of LUTs)	Cameron Estimation (# of LUTs)	% Error
AddD	1124	1150	2.31
AddM	879	917	4.32
Convolution	1783	1860	4.32
Convolution5	3235	3269	1.05
Convolutionsm	2609	2616	0.27
Dilation	1365	1451	6.30
Erosion	1373	1451	5.68
Gaussianfilter	1117	1125	0.71
Laplacefilter3 × 3	1198	1247	4.09
Max	821	832	1.34
Maxfilter	1407	1436	2.06
Min	867	832	4.04
Minfilter	1230	1200	2.44
Mp4	1150	1154	0.35
MultiplyM	920	958	4.13
MultiplyD	1171	1182	0.94
Prewitt	1083	1020	5.82
Prewittmag	1815	1792	1.27
Reduce	1013	1016	0.30
Robertsmag	1441	1414	1.87
Sobelmag	1943	1921	1.13
Sqrt	984	944	4.07
SubtractD	1121	1123	0.18
SubtractM	865	901	4.16
Threshold	911	903	0.88
Average Error: 2.56%		Weighted Average Error: 0.86%	

Table V. Benchmark Results

Benchmark	Synplify Estimation (# of LUTs)	Cameron Estimation (# of LUTs)	% Error
Open	4500	4780	6.22
Close	4500	4780	6.22
Wavelet	2172	2065	4.93
Tridiagonal	7476	7188	3.85
Average Error: 5.30%		Weighted Average Error: 0.88%	

—The *tridiagonal* code solves the matrix equation [A] • [X] = [B]. The basic idea is to use the tridiagonal matrix A[8,8] as input, calculate [B] and then solve [A] • [X] = [B] iteratively to obtain [X].

Open is composed of erosion operation followed by dilation, but when the two loops are fused, it runs twice as fast as the individual routines, at the expense of more area. In wavelet and tridiagonal, the loops are fully unrolled to exploit more parallelism. Table V shows the results of the estimation tool versus synthesis. This is a challenging problem; nonetheless our estimator comes within about 6% of the actual space used by the design.

Table VI shows timing results; the synthesis tool takes 6.2 minutes on an average to do the synthesis and mapping. Our estimator takes only 1 millisecond to do the estimation.

Table VI. Comparison of Estimation Time

Benchmark	Synthesis Time	Cameron Estimation Time
Open	3.4 mins	1 millisec
Close	3.4 mins	1 millisec
Wavelet	8.3 mins	1 millisec
Tridiagonal	10.0 mins	1 millisec
Average	6.2 mins	1 millisec

```
uint20 [:,:] main (uint8 image[:,:], uint8 kernel[:,:])
{
 uint20 res[:,:] =
          // PRAGMA (strip-mine(4,3))
          for window win[3,3] in image
          { uint20 val =
                 for elem1 in win dot elem2 in kernel
                     return(sum((uint20)elem1*elem2));
          } return(array(val));
} return (res);
```

Fig. 12. Convolution.

5.3 Effect of SA-C Compiler Optimizations

This final set of experiments demonstrates the performance of the estimation tool in more complicated algorithms. These are representative of 'real-world' applications that the SA-C compiler may be called upon to implement. Estimation is tricky in this scenario, since the final structure of the DFG produced by the compiler is significantly different from the original program, due to the extensive optimizations performed by SA-C.

Example 1. The first example examines the effect of strip-mining on the resource usage of the design. We use the *convolution* routine shown in Figure 12, which does a 3×3 convolution of an *image* with *kernel*. We do estimation of this SA-C code for the default case (no strip-mining) and then strip-mine the loop for windows of sizes [4,3], [5,3], [6,3], [7,3], [8,3] and [20,3] (indicated via a PRAGMA[4]).

Table VII shows the resource usage using the various strip-mining choices. Loop strip-mining followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. In this example, strip-mining reduces the total number of consecutive iterations by exploiting loop level parallelism, at the cost of larger area. An application programmer can quickly use this estimation to determine the size of the window, and hence the degree of parallelism, that can be used in strip-mining the inner loop body. Without fast estimation, it would take hours, if not days, to test each design.

Another important property of the estimation techniques is *fidelity*. The estimator's fidelity is highest if, for a single design that has had been optimized in several ways, or for several different designs, it predicts a ranking of the resource usage of the designs that is in the same order as that which occurs

[4]PRAGMAs in SA-C source code allow control of individual optimizations.

Table VII. Strip-Mining Results

Convolution	Synplify Estimate (# of LUTs)	Cameron Estimate (# of LUTs)	% Error	% LUT used on Virtex XCV1000
No strip-mining	1783	1949	9.3	7.04
Strip-mine(4,3)	2609	2794	7.09	9.43
Strip-mine(5,3)	3458	3516	1.67	12.71
Strip-mine(6,3)	4269	4134	3.16	14.95
Strip-mine(7,3)	5122	4980	2.77	18.01
Strip-mine(8,3)	6709	5686	4.5	20.56
Strip-mine(20,3)	15883	14727	7.2	53.26
Average Error: 5.09%		Weighted Average Error: 5.26%		

```
// Prewitt
int8 V[3,3] = { {-1, -1, -1}, { 0,  0,  0}, { 1,  1,  1} };
int8 H[3,3] = { {-1,  0,  1}, {-1,  0,  1}, {-1,  0,  1} };

uint8 R[:,:] = for window W[3,3] in Image {
        int8 iph, int8 ipv =
                for h in H  dot  w in W  dot v in V
                        return(sum(h*w), sum(v*w));

        uint8 mag = sqrt(iph*iph + ipv*ipv);
} return( array(mag) );

// Threshold
uint8 T[:,:] = for pix in R{
        uint8 t = pix>127 ?255 : 0;
} return(array(t));
```

Fig. 13. Prewitt and threshold (two independent loops).

with the actual synthesis. In other words, even when the estimator's *absolute* estimates are inaccurate, the estimator still produces meaningful results if the *relative* accuracies are consistent with one another. Fidelity is important in iterative design optimizations since it provides the correct guidance to the optimization tasks. The results reported in Table VII show that the estimator correctly predicts the relative impact of design optimization steps.

Example 2. In this example, we present the effect of both loop fusion and strip-mining on a combination of *prewitt* and *threshold*. We examine the estimation reported for three cases: (a) independent loops, (b) loop fusion, and (c) a combination of strip-mine and loop fusion, for the SA-C code shown in Figure 13. When no loop fusion is applied, two loops run on the reconfigurable board, one of them being activated multiple times.

The performance of many systems is often limited by the time required to move data to the coprocessor. Loop fusion is helpful in this case, since it eliminates intermediate data storage and therefore reduces data traffic. Figure 14 shows the two loops after they are manually unrolled and fused; the equivalent

```
// Loops fused
uint8 T[:,:] = for window W[3,3] in Image {
        int8  iph  =    (W[0,2]+W[1,2]+W[2,2])  -
(W[0,0]+W[1,0]+W[2,0]);
        int8  ipv  =    (W[2,0]+W[2,1]+W[2,2])  -
(W[0,0]+W[0,1]+W[0,2]);

        uint8 mag = sqrt(iph*iph + ipv*ipv);
        uint8 t = mag>127 ? 255 : 0;
} return( array(t) );
```

Fig. 14. Prewitt + Threshold (loops fused).

```
// Loop strip-mining
uint8 T[:,:] = for window W[4,3] in Image step(2,1) {
        int8 iph1 =  (W[0,2]+W[1,2]+W[2,2]) - (W[0,0]+W[1,0]+W[2,0]);
        int8 ipv1 =  (W[2,0]+W[2,1]+W[2,2]) - (W[0,0]+W[0,1]+W[0,2]);
        uint8 mag1 = sqrt(iph1*iph1 + ipv1*ipv1);
        uint8 t1 = mag1>127 ? 255 : 0;
        int8 iph2 =  (W[1,2]+W[2,2]+W[3,2]) - (W[1,0]+W[2,0]+W[3,0]);
        int8 ipv2 =  (W[3,0]+W[3,1]+W[3,2]) - (W[1,0]+W[1,1]+W[1,2]);
        uint8 mag2 = sqrt(iph2*iph2 + ipv2*ipv2);
        uint8 t2 = mag2>127 ? 255 : 0;
        uint8 t[2,1] = {{t1},{t2}};
} return( tile(t) );
```

Fig. 15. Prewitt and threshold (loops strip-mined).

of this optimization can be activated in SA-C by a pragma. The result is only one loop running on the reconfigurable board that is activated only once. Loop fusion sometimes does redundant computation. If FPGA space is plentiful, this is not a problem since the computation is done in parallel. If space is scarce, then other optimizations can be applied to remove the redundancies.

Loop strip-mining followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. Figure 15 shows the result of manually strip-mining the loops. This optimization wraps the original loop inside a new loop with a 4×3 window generator. Strip-mining results in a single loop running on the reconfigurable board that requires only half the number of iterations than in the earlier case. Thus, new loops are created during optimization, and the resource usage is affected depending on the window size.

Table VIII shows the estimation results (per loop iteration). When the two loops are independent, 1644 LUTs are used. Fusing the two loops reduces this to 1063 LUTs and improves execution performance, since fewer iterations are now required.

This example is a small piece of code that appears in the larger tridiagonal algorithm; compile-time estimation becomes more crucial in larger algorithms. They might contain hundreds of loops and a large number of optimization combinations can be performed on them. Quick estimations are necessary in order for the programmer to search for the best solution.

Table VIII. Prewitt +Threshold Area Estimation

Type of Optimization on Prewitt + Threshold	Estimated LUT Usage	% Area on XCV1000
Independent Loops	1644	1.28
Loops fusion (default)	1063	0.83
Loop fusion + Strip-mine(4, 3)	1280	1.00

In the case of large programs, we have had reasonable success in using the speed vs. space optimizations, up to around 75% FPGA utilization. This success is due partly to our original “tweaking” of the VHDL code that implements the DFG nodes—some implementations appear to compile more reliably than others with Synplify.

The estimator can also be used in a “hybrid” approach, combining both estimator and actual synthesis results. Using the techniques described in Xu and Kurdahi [1996] and Ohm et al. [1994, 1995], more accurate estimation of timing can be applied at the “outer loop” of the compilation procedure after a design has been synthesized. This requires performing a synthesis step, but only on the relatively simpler outer loop.

6. PREVIOUS WORK

Xu and Kurdahi [1996] have developed a strategy for accurate prediction of quality metrics for FPGA based designs. Given a netlist description, the tool estimates the area of the design in terms of configurable logic blocks. The FPGA timing estimator uses placement information and area approximation to estimate the timing. The average estimation error in delay in this scheme is about 5.3%, while the worst-case error is 13.2%. Moreover, the experiment takes 1 to 2 orders of magnitude less time to compute than the synthesis tools.

The idea of lower bound area estimation from behavioral descriptions for multiplexer-based and bus-based architectures is explored in Ohm et al. [1994, 1995]. The behavioral description is expressed as a dataflow graph, and the total performance and clock period expressed in real time are given as constraints. Given all this information the tool estimates the lower bounds on the number of functional units of each type, the number of registers and the number of buses.

A power estimation approach for SRAM-based FPGAs is discussed in Weis et al. [2000]. The authors infer that if finite state machines dominate the structure, then the power estimation approach can be successfully applied. If the structure has more combinational logic, then the algorithm fails.

In Kannan et al. [2002], the authors propose a uniform reporting metric for routability estimation. Although interconnect management is an important issue, we consider it to be orthogonal to our work.

Estimating area and performance for FPGAs at DFG level is explored in Enzler et al. [2000]. The presented approach is however for static designs only.

Although the results presented in Shayee et al. [2003] are preliminary, the authors seem to explore an approach that is very similar to our overall framework. The authors examine the effect of various program transformations on the I/O resources available on the FPGA.

7. CONCLUSION

In this article, we have presented a compile-time area estimation approach for LUT-based FPGAs. The estimation model was developed to aid the SA-C programmer in selecting optimizations that affect the resource usage of a SA-C program. Our estimation approach operates at a much higher level—DFG rather than the netlist representation. The algorithms we use are not as complex as the ones used in the commercial synthesis tools, but the estimates can be found must faster. We have successfully demonstrated our estimation technique on a variety of both simple and more complex benchmarks, with experimental results indicating that our technique achieves an accuracy within around 2.5% for small image-processing operators and 5.0% for larger benchmarks. The worst-case error we have observed is slightly over 10%. Importantly, our estimator achieves good fidelity, or relative prediction accuracy. The time required for our estimates is no more than milliseconds, as compared to minutes for a synthesis tool, thereby providing the SA-C programmer with the information needed to try numerous optimizations on a design. Even though the estimates that our tool computes are specific to the Xilinx Virtex (XCV1000) FPGA, it can be easily modified to work for a variety of other FPGAs.

REFERENCES

ANNAPOLIS MICRO SYSTEMS, INC. 2000. *WILDSTAR Reference Manual.* Annapolis Micro Systems, Inc., Annapolis, MD. www.annapmicro.com.

BÖHM, W., BEVERIDGE, R., DRAPER, B., ROSS, C., CHAWATHE, M., AND NAJJAR, W. 2002a. Compiling ATR probing codes for execution on FPGA hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE Computer Society Press, Los Alamitos, CA. (Napa Valley, CA). 301–302.

BÖHM, W., DRAPER, B., NAJJAR, W., HAMMES, J., RINKER, R., CHAWATHE, M., AND ROSS, C. 2001. One-step compilation on image processing applications to FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Rohnert Park, CA). IEEE Computer Society Press, Los Alamitos, CA.

BÖHM, W., HAMMES, J., DRAPER, B., CHAWATHE, M., ROSS, C., RINKEH, R., AND NAJJAR, W. 2002b. Mapping a single assignment programming language to reconfigurable systems. *Super-computing 21*, 117–130.

COHEN, A., DAUBECHIES, I., AND FEAUVEAU, J. 1992. Bi-orthogonal bases of compactly supported wavelets. In *Commun. Pure Appl. Math. XLV*, 485–560.

DEHON, A. 2000. The density advantage of reconfigurable computing. *IEEE Comput. 33*, 4, 41–49.

DRAPER, B., BÖHM, W., HAMMES, J., NAJJAR, W., BEVERIDGE, R., ROSS, C., CHAWATHE, M., DESAI, M., AND BINS, J. 2001. Compiling SA-C programs to FPGAs: Performance results. In *Proceedings of the International Conference on Vision Systems* (Vancouver, B. C., Canada). 220–235.

ENZLER, R., JEGER, T., COTTET, D., AND TRSTER, G. 2000. High-level area and performance estimation of hardware building blocks on FPGAs. In *Proceedings of the Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications.* 525–534.

HAMMES, J., BÖHM, W., ROSS, C., CHAWATHE, M., DRAPER, B., AND NAJJAR, W. 2001a. High performance image processing on FPGAs. In *Proceedings of the Los Alamos Computer Science Institute Symposium.* Santa Fe, NM.

HAMMES, J., BÖHM, W., ROSS, C., CHAWATHE, M., DRAPER, B., RINKER, R., AND NAJJAR, W. 2001b. Loop fusion and temporal common subexpression elimination in window-based loops. In *Proceedings of the IPDPS 8th Reconfigurable Architectures Workshop* (San Francisco, CA).

KANNAN, P., BALACHANDRAN, S., AND BHATIA, D. 2002. On metrics for comparing routability estimation methods for FPGAs. In *Proceedings of the 39th Conference on Design Automation* (New Orleans, LA).

KULKARNI, D., NAJJAR, W., RINKER, R., AND KURHADI, F. 2002. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom. Computing Machines (FCCM)* (Napa Valley, CA). IEEE Computer Society Press, Los Alamitos, CA.

NAJJAR, W. A., BÖHM, W., DRAPER, B., HAMMES, J., RINKER, R., BEVERIDGE, R., CHAWATHE, M., AND ROSS, C. 2003. From algorithms to hardware—A high-level language abstraction for reconflgurable computing. *IEEE Comput. 36*, 8 (Aug.), 63–69.

OHM, S., KURDAHI, F., AND DUTT, N. 1994. Comprehensive lower bound estimation from behavioral descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Jose, CA). IEEE Computer Society Press, Los Alamitos, CA. I82–189.

OHM. S., KURDAHI, F., DUTT, N., AND XU, M. 1995. A comprehensive estimation technique for high-level synthesis. In *Proceedings of the International Symposium on System Synthesis (ISSS).*

RINKER, R., CARTEH, M., PATEL, A., CHAWATHE, M., ROSS, C., HAMMES, J., NAJJAR, W., AND BÖHM, W. 2001. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. VLSI Design 9*, 130–139.

SHAYEE, K., PARK, J., AND DINIZ, P. 2003. Performance and area modeling of complete FPGA designs in the presence of loop transformations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Napa, CA). IEEE Computer Society Press, Los Alamitos, CA.

WEIS, K., OETKER, C., KATOHAN, I., STECKSTOR, T., AND ROSENSTIEL, W. 2000. Power estimation approach for SRAM-based FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Logic and Applications (FPGA 2000)* (Monterey, CA). IEEE Computer Society Press, Los Alamitos, CA, 195–202.

XILINX INC. 2000. *Virtex 2.5V Field Progammable Gate Array.* Xilinx, Inc. www.xilinx.com.

XU M. AND KURDAHI, F. 1996. Accurate prediction of quality metrics for logic level designs targeted towards lookup table based FPGAs. *IEEE Trans. VLSI Systems*.

Received January 2003; revised April 2004 and February 2005; accepted June 2005