

# Fast Compilation for Pipelined Reconfigurable Fabrics

Mihai Budiu and Seth Copen Goldstein  
mihai@cs.cmu.edu, seth@cs.cmu.edu  
Carnegie Mellon University

## Abstract

In this paper we describe a compiler which quickly synthesizes high quality pipelined datapaths for pipelined reconfigurable devices. The compiler uses the same internal representation to perform synthesis, module generation, optimization, and place and route. The core of the compiler is a linear time place and route algorithm more than two orders of magnitude faster than traditional CAD tools. The key behind our approach is that we never backtrack, rip-up, or re-route. Instead, the graph representing the computation is preprocessed to guarantee routability by inserting lazy noops. The preprocessing steps provides enough information to make a greedy strategy feasible. The compilation speed is approximately 3000 bit-operations/second (on a PII/400Mhz) for a wide range of applications. The hardware utilization averages 60% on the target device, PipeRench.

## 1 Introduction

Reconfigurable computing has the potential to change the way computing is performed. Such systems offer a many-fold increase in performance by better utilizing the large amounts of silicon available to chip designers, particularly for applications which can be turned into pipelined datapaths. However, it is notoriously hard to compile to such systems. In this paper we describe a compiler which aims to solve this problem. The compiler uses a natural dataflow representation of the program to create pipelined datapaths two orders of magnitude faster than traditional tools.

The source language for the compiler, DIL, is a single assignment language that can be easily used by either programmers or as an intermediate language in a high-level language compiler. DIL is expressive enough that it is used to create parameterized modules which form part of the compilation process.

The current compiler targets pipelined reconfigurable architectures like PipeRench, although our techniques are gen-

eral enough to apply (at least in part) to other fabrics, such as commercial FPGAs. For PipeRench the compiler achieves excellent compilation speeds (approximately 3000 bit-operations/second) and yields good hardware utilization (over 60%). The key to the speed is a deterministic, linear-time place and route algorithm, which uses the novel technique of lazy noops. Place and route itself runs at 6000 bit-operations/second. An important reduction in complexity comes from the fact that, instead of creating a flat netlist and then recreating the datapath implied by the source program, the DIL compiler maintains the structure of the program throughout the compilation process.

In the next section we describe DIL, our data-flow intermediate language which is the input for our compiler. Section 2 describes the current class of architectures that our compiler targets. We describe the language and some of the important passes of the compiler in Section 3. In Section 4 we describe our linear time place and route algorithm. In Section 5 we evaluate the compiler's performance on a series of benchmark kernels. We cover related work in Section 6 and conclude in Section 7.

## 2 PipeRench

The current version of the DIL compiler targets PipeRench, an instance of the class of pipelined reconfigurable fabrics [23]. From the compiler's point of view the two most important characteristics of PipeRench are that it (1) supports hardware virtualization and (2) is optimized to create pipelined datapaths for word-based computations. Hardware virtualization allows PipeRench to efficiently execute configurations larger than the physical fabric, which relieves the compiler from the onerous task of fitting the configuration into a fixed size fabric. PipeRench achieves hardware virtualization by structuring the fabric (and configurations) into pipeline stages, or *stripes*. The stripes in an application are time multiplexed onto the physical stripes (see Figure 1). This requires that every physical stripe be identical. It also restricts the computations it can support to those in which the state in any pipeline stage is a function of the current state of that stage and the current state of the previous stage in the pipeline. In other words, the dataflow graph of the computation cannot have long cycles.

Each stripe in PipeRench is composed of  $N$  processing elements (PEs). In turn, each PE is composed of  $B$  identically configured 3-LUTs,  $P$   $B$ -bit pass registers, and some control

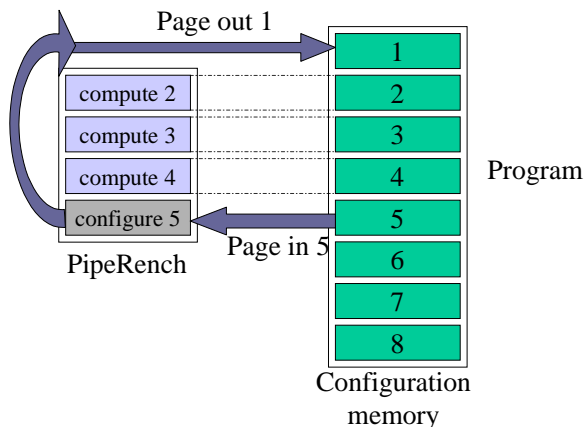


Figure 1: *Hardware virtualization in PipeRench overlaps computation with reconfiguration and provides the illusion of unlimited hardware resources.*

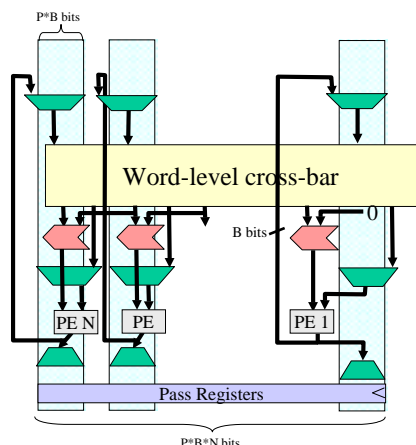


Figure 2: *The interconnection network between two adjacent stripes. All switching is done at the word level. All thick arrows denote  $B$ -bit wide connections.*

logic. The three inputs to the LUTs are divided into two data inputs (A and B) and a control input similar to [7]. Each stripe has an associated *inter-stripe interconnect* used to route values to the next stripe and also to route values to other PEs in the same stripe. An additional interconnect, the *pass-register interconnect*, allows the values of all the pass registers to be transferred to the pass registers of the PE in the same column of the next stripe.

The structure of the interconnect is depicted in Figures 2 and 3. Both the inter-stripe interconnect and the pass-register interconnect switch  $B$ -bit wide buses, not individual bits. A limited set of bit permutations are supported in the interconnect by barrel shifters which can left shift any input coming from the inter-stripe interconnect. Currently, the inter-stripe interconnect is implemented as a full crossbar.

The richness of the network is, however, misleading for two reasons: we only have limited access to the interconnect and the fact that all switching is bus based. All values which

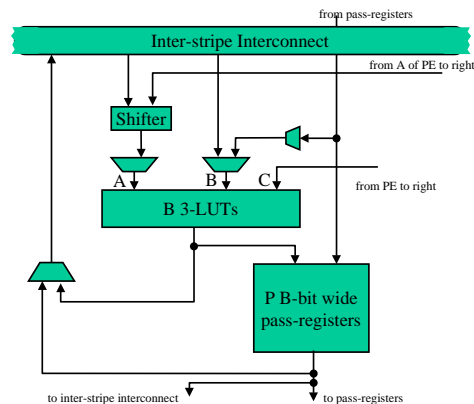


Figure 3: *The structure of a processing element. There are  $N$  PEs in each stripe. Details about the zero-detect logic, the fast carry chain and other circuitry are left out.*

are destined to a non-adjacent stripe must be placed in a pass register. Furthermore, access to the inter-stripe interconnect is limited to either the output of the LUTs or one of the  $P$  pass registers. The only place where a bit can change its position in a word is in the barrel shifters. The barrel shifters provide a limited range of movements for the bits, so some kinds of wiring patterns in the circuit require several stages through the interconnection network (for instance, the bit-reversal of all the bits in a PE).

DIL is parameterized to allow the generations of configurations for chips with any number of PEs ( $N$ ), pass registers ( $P$ ), and PE granularity ( $B$ ). It can also be parameterized by the connectivity of the inter-stripe interconnect. In this paper we report results for stripes with 16 8-bit PEs each having 8 pass registers. The inter-stripe interconnect is a full crossbar, which has  $N$  inputs and  $2N$  outputs, i.e., the A and B inputs to the PE can each have a different source.

### 3 The DIL Language and Compiler

DIL is designed to be both the target of a high-level language compiler (e.g., C or Java) and a language that a programmer can use to describe algorithms for reconfigurable computing systems. DIL does not expose any of the peculiarities of the fabric, being completely hardware independent. It provides a rich set of operators (the set of operators of the C language, augmented with a few other).

Our thesis is that for widespread acceptance of reconfigurable architectures, programmer effort must be substantially reduced. Thus our main goal is to provide a fast compiler for a high-level language. We trade solution quality for speed, which enables us to perform very fast compilation, compared to classical CAD tools.

#### 3.1 The DIL Language

DIL is a high-level language. There is no notion of hardware resources, timing, or physical layout for the DIL programmer. As a consequence most of the compilation process (including

```

1
2  typedef fixed<*,0> uint;           // arbitrary width
3
4  // one multiply-accumulate step
5  tap(uint xin in, uint xout out, uint yin in, uint yout out, uint w)
6  {
7      yout = yin + xin*w;
8      xout <1= xin;                 // delay by one register element
9  }
10
11 // Generic FIR filter; weights and # of taps are passed as arguments
12 filter(uint xin in, uint yout out, uint weights, uint taps)
13 {
14     uint x[taps], y[taps];        // intermediate results
15     uint i;
16
17     tap(xin, x[0], 0, y[0], weights[0]); // the first tap
18     for (i=1; i < taps; i=i+1) {     // the rest of the taps
19         tap(x[i-1], x[i], y[i-1], y[i], weights[i]);
20     }
21     yout = y[taps-1];             // assign the result to the output
22 }
23
24 // Instantiate one FIR filter with 5 taps
25 main(fixed<8,0> xin in, fixed<12,0> yout out)
26 {
27     uint taps = 5;
28     uint w[taps] = { 5,4,3,2,1 };
29
30     filter(xin, yout, w, taps);
31 }

```

Figure 4: DIL code to implement a 5 tap FIR filter. `main` is the function used to interface to the outside world.

code optimizations) should be portable to architectures other than PipeRench. DIL is a single assignment language: any variable can be assigned to only once. With few exceptions it is very similar to Silage and behavioral Verilog.

DIL allows the programmer to manipulate arbitrary-width integer values and, unless directed explicitly, ensures that no information loss occurs due to overflow or conversions. A powerful module system and a parameterized module library allow the user to customize the meaning of the language’s operators.

We illustrate the features of DIL using the example in Figure 4 which shows how a Finite Impulse Response filter (FIR) with 5 taps could be implemented. The definition of the circuit is given by  $Output_i = \sum_{j=0}^{taps} Input_j \cdot weight_{taps-j}$ . The circuit described by this program will accept one input and will produce one output every clock cycle. The latency of the circuit will be at least 5 clock cycles, but possibly more due to the insertion of additional pipeline stages.

The FIR code in Figure 4 contains most of the important features of DIL.

- The type declaration in Line 2 declares the type `uint` to be an unsigned integer of arbitrary width (the ‘\*’ denotes an arbitrary width). Each `uint` variable in the program will actually have a different width, which is determined by the compiler. While any variable in the program may be given an explicit width, the only variables which must have their widths specified are the `in` and `out` parameters to `main`.
- DIL features procedure-like modules (e.g., `tap` in Line 5).

One can pass (or return) arrays to (or from) modules. For example, `weights` in Line 12 is an array of constants. Array elements, like all variables, can be assigned to only once.

- The symbol ‘<1=’ (e.g., in Line 8) stands for “delayed assignment with one unit of delay.” The final circuit will have 5 registers which are used to hold the last 5 input values for the convolution computation.
- The `for` construct (on Line 18) is used to instantiate an entire series of taps, each with a different weight. As in Verilog, the `for` loops are unrolled at compile-time.
- All of the C operators are available, including integer multiplication and division. We have also added some other useful operations, like bit-range selection (denoted by `value[high, low]`) and bit-string concatenation (denoted by `x.y`).

### 3.2 The DIL Compiler

The whole compilation process is depicted in Figure 5. The compiler first reads in the architecture of the target circuit. For example, with PipeRench this file describes the width of the processing elements ( $B$ ), the number of I/O buses available, the signal propagation delays, the number of pass registers ( $P$ ), the structure of the inter-stripe interconnect, the target clock speed (i.e. the critical path), etc.

In the evaluation phase the compiler inlines all modules, unrolls all loops and generates a straight-line, single assignment

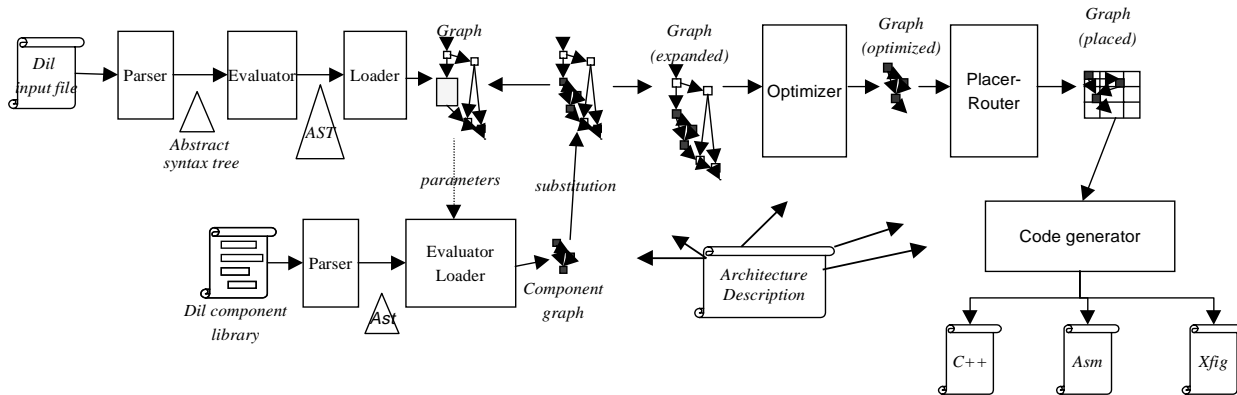


Figure 5: The main stages of the DIL compiler.

```

1
2  main(fixed<8,0> xin in, fixed<12,0> yout out)
3  {
4      fixed<*,0> x[5], y[5];
5
6      y[0] = 0 + xin *5;    x[0] <1= xin;
7      y[1] = y[0] + x[0]*4; x[1] <1= x[0];
8      y[2] = y[1] + x[1]*3; x[2] <1= x[1];
9      y[3] = y[2] + x[2]*2; x[3] <1= x[2];
10     y[4] = y[3] + x[3]*1; x[4] <1= x[3];
11     yout = y[4];
12 }

```

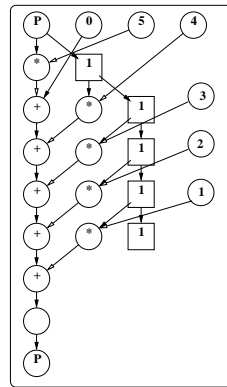


Figure 6: The straight-line program and internal graph representation of the FIR filter.

program. The result obtained from the FIR filter is shown at the left of Figure 6.

After this, the program is converted into a graph upon which the rest of compiler operates. The right side of Figure 6 shows the graph synthesized for the filter. The nodes of the graph are operators and the edges represent wires. The P nodes represent input-output ports, the square boxes are the delay registers.

The graph structure is hierarchical, in the sense that the nodes can be graphs themselves (as in Figure 9). The hierarchical representation is a direct result of our node synthesis methods. For instance the graph in Figure 9 has been created by substituting for each multiplier node in Figure 6 a corresponding multiplier implementation. The hierarchical representation is particularly useful when we wish to avoid processing a group of nodes. We group the nodes into a subgraph and tag the subgraph so that it is viewed as a single node.

The DIL compiler is implemented in 29000 lines of C++ code. Nodes, graphs and wires are C++ objects. The compiler is extremely modular: inserting a new compiler pass means writing a new method for the graph class. Right now the compiler consists of more than 30 passes, most of which run in linear time in the size of the graph. In the rest of this section we describe some of the more interesting passes and their effects on the FIR program. Section 4 is devoted to the description of the fast place and route algorithm.

### 3.2.1 Type and Value Inference

This pass computes the minimum width required for the wires (and implicitly the amount of logic required for computations). The compiler determines the minimum widths necessary for each wire by symbolically evaluating the graph over the values  $\{0, 1, x\}$ , where  $x$  represents the unknown value. In addition to storing a bit vector with the best known value for each wire, the compiler also stores the maximum absolute value that can be on the wire. This is used to restrict the wire widths from growing unnecessarily for addition. For example, if  $a, b,$  and  $c$  are 4-bit unknown wires, then  $d=a+b; e=d+c$  yields a maximum width of 5 bits, not the 6 bits which would result if no maximum were kept.<sup>1</sup>

Additionally, if the compiler can determine that any of the bits on a wire are constant, it will use those constants to help simplify the graph. For example, this pass would determine that the assignment  $a = (b \& 5) [1, 1]$  causes  $a$  to be assigned the single 1-bit wide constant 0.

### 3.2.2 Optimizations

The compiler applies a whole set of optimizations to the graph, some typical in traditional CPU compilers and some more

<sup>1</sup>In all the graphs in the following figures the number printed close to each wire indicates the compiler-derived width of the wire.

```

1  typedef unsigned fixed<*,0>;
2
3  mult(unsigned input0 in, unsigned input1 in, unsigned output out)
4  {
5      unsigned partial[input1_size + 1];
6      unsigned i;
7
8      partial[0] = 0;
9      for (i=1; i<=input1_size; i=i+1) {
10         partial[i] = partial[i-1] + ((input1[i-1,i-1]) ? (input0<<(i-1)) : 0);
11     }
12     output = partial[input1_size];
13 }

```

Figure 7: A piece of the library module written in DIL which implements unsigned multiplication as an array multiplier.

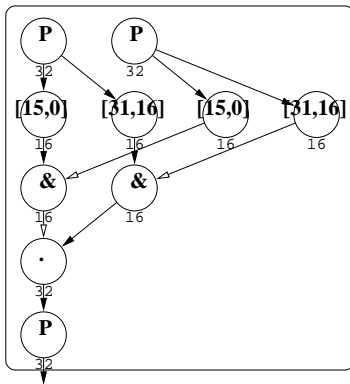


Figure 8: Decomposition of a large “and” gate.

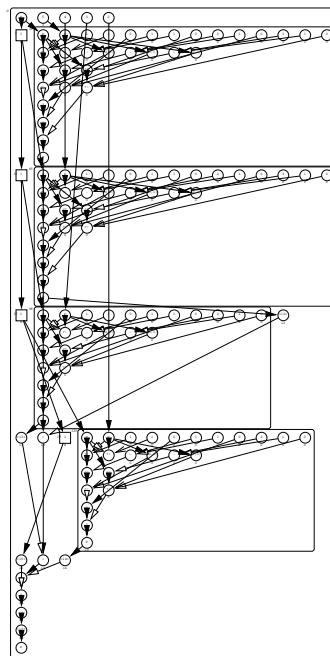


Figure 9: The FIR filter with multipliers substituted.

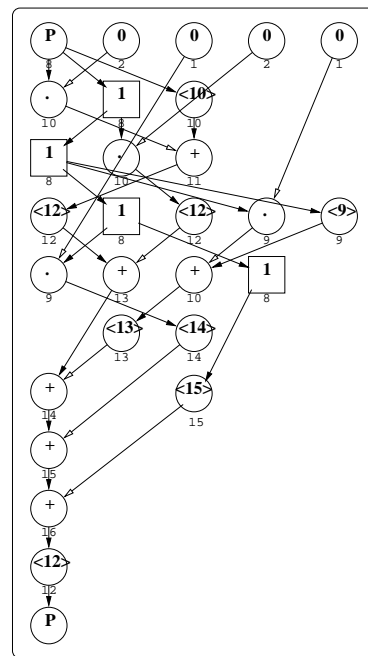


Figure 10: The FIR filter after optimizations.

characteristic of CAD tools. The optimization passes include: common sub-expression elimination, dead code elimination, bit-level constant propagation, algebraic simplifications (e.g.  $a^a = 0$ ), register reduction (this is a form of re-timing [19]), and interconnection simplification (e.g. if  $a$  and  $b$  are 16-bit quantities, the expression  $(a \cdot b)[7, 0]$  is actually equivalent to  $b[7, 0]$ ).

### 3.2.3 Library loading

The library is a powerful element for extensibility, retargetability and customization. The library is similar in power to module generators in [8, 20, 4]. The library modules are more powerful than those in [22, 18, 14] because the library modules

are optimized by the compiler when instantiated. Currently we use the library to implement operators which cannot be synthesized directly in hardware (e.g.  $*$ ,  $\%$ ) and to decompose operations on wide wires into smaller pieces. Figure 8 shows how a “bitwise and” operation on 32 bit values is decomposed into narrower pieces.

An example of a (slightly simplified) library module to implement unsigned multiplication is shown in Figure 7. When the compiler encounters a multiplication operation in the compiled program, it initializes the variables `input0`, `input1`, `input1_size`, etc., and attempts to load the module `mult` from the library. The library is a plain text DIL program. The programmer can rewrite the above module, for instance as a Booth multiplier or as a CSD multiplier.

Figure 9 shows (in a very low resolution) how the FIR filter looks after the multipliers are decomposed using the library. Figure 10 shows how FIR looks after the optimization passes have been run. Because all multiplicands are manifest constants they are successfully reduced to a few adders (only the adders corresponding to the 1 bits in the multiplicand are retained). The nodes labeled  $\langle x \rangle$  are width-conversion operators, which either truncate or sign-extend the value to  $x$  bits.

### 3.3 Width adjustment

The width adjustment pass adjusts the widths of the wires to match the width of the PE, i.e., it makes all wires multiples of  $B$  bits. While this may seem trivial at first, it proved to be very tedious due to the nature of 2's complement arithmetic. The obvious solution, to sign-extend all magnitudes up to the next multiple of  $B$  bits, is very costly because sign extension consumes precious circuitry (2 PEs per value in the current implementation). We have implemented a solution which relies on the fact that adjusting the sources of an operation most often will cause the output to be adjusted too. For example, adding two 2-bit wires yields a 3-bit wire, but if the 2-bit wires are correctly sign extended, then the result is also already the correct size.

However the complete picture is more complex, as sign-extension is not always the correct way to bring a value to a certain size. For example, suppose  $a, b$ , are each 2-bit wide signed values and a PE is 4 bits wide. If  $a$  and  $b$  are compared with a  $<$  operator, then to get a correct result from a 4-bit comparison they should both be sign-extended. On the other hand, if we want to compute  $a . b$  (the concatenation), it turns out that this is most efficiently implemented on PipeRench (for this width combination) as the operation  $(a \ll 2) | b$ , which requires  $b$  to be padded with zeros instead of being sign-extended (if  $b$  were negative, sign-extending it would give the wrong result).

#### 3.3.1 Code generation

The normal output of the compiler is “assembly language”, essentially a description of a fully placed and routed netlist. Currently we generate assembly for the only architecture supported, PipeRench. Porting to a new architecture would involve writing a pass to translate an (annotated) graph to the suitable output and possibly changing the library and architecture input file.

For debugging support (both for debugging the compiler and DIL programs) the compiler also outputs a graphical representation of the circuit in the language of the Unix XFIG drawing utility (all the circuit graphs in this paper have been generated automatically in this way).

The compiler can also generate a C program which simulates the circuit. (Alternatively, the assembler can generate behavioral Verilog or a text file which can be used by a Java-based visual simulator.)

## 4 Place and Route

The core of the DIL compiler is its place and route (P&R) algorithm. The P&R algorithm is a deterministic, linear-time, greedy algorithm. The measured performance, as shown in

Section 5, is around 6000 bit-operations/second on a Pentium II at 400Mhz, or about three orders of magnitude faster than commercial tools.

The key idea behind our P&R algorithm is to prepare the graph to ensure routability by decomposing complex interconnection patterns into sequences of simpler ones. The decomposition is performed by inserting noops<sup>2</sup> in the graph. The insertion of noops is sufficient by not necessary to guarantee routability. This allows us to treat the noops as *lazy noops*. A lazy noop is only instantiating when the P&R routine cannot make progress.

There are several reasons why our P&R algorithm is so fast. Some of them are specific to PipeRench and some applicable to other fabrics. In particular, we capitalize on the structure of the graphs created by computations. In the rest of this section we discuss the reasons why our algorithm performs well in spite of the fact that it is deterministic and then we describe the details of the algorithm.

### 4.1 Why Our P&R Is Fast

PipeRench is a “compiler friendly” fabric; here are some of its features which make P&R a simpler task:

- The hardware resources are virtualized. The compiler does not have to fit the circuit in the available hardware, so it can indeed trade utilization for compilation speed.
- The interconnection network is relatively rich. The crossbar between two stripes allows values to be routed between any two vertical columns. This enables high quality solutions and trims down the search space, because there are many feasible solutions (i.e. we can connect two operators placed in any two columns).
- The very wide pass-register bus allows us to propagate values between two remote stripes without consuming any processing elements for this purpose.
- The interconnection network is homogeneous: between any two stripes the network is identical. (This feature is not essential, but it simplifies the algorithms.)

There are however other design decisions which we made which keep the algorithm target-independent and which make P&R fast:

- We place operators at very high granularities. For instance, we place a 32-bit adder as a single entity (if allowed by the propagation delay of the carry chain), instead of flattening it to gate level and manipulating hundreds of gates individually. This provides at least one order of magnitude reduction in complexity.
- We compile a restricted language, which can only describe unidirectional data flow (i.e. pipelined combinatorial circuits). This simplifies the routing, because by traversing the graph in topological order we always have one degree of freedom in the placement of the wires (i.e. we can choose where to place one end).
- The greedy, non-backtracking algorithm is inherently fast. In order to enable a greedy placement, we have to make sure that once a node is placed there still exists

<sup>2</sup>In actual fact, what we call a noop is really a pass operator: the output gets the input value.

a solution for the placement of the remaining nodes. This is summarized by the rule: “no matter what the placement of the nodes so far, we can always make further progress by placing at least one other node.” The global analysis and the technique of lazy noops, presented shortly, ensure that this rule is always true.

#### 4.2 Why P&R Is Still Hard on PipeRench

There are several features of the PipeRench interconnect which make P&R hard and which are also present in varying degrees in commercial FPGAs.

- The width of the stripes and of the pass-register bus are limited. These are hard limits which have to be managed by the compiler.
- The input ports of the crossbar are scarce, as depicted in Figure 2. Although we have a crossbar to switch the  $N$  busses in arbitrary fashions, many more values ( $N \times (P + 1)$ ) compete for the inputs of the crossbar. More than that, each input port of the crossbar services exactly  $P + 1$  fixed lines:  $P$  from the pass-register bus and one from the feedback loop; no more than one of these  $P + 1$  values can enter the crossbar at any time. The placement of values in the pass-registers is thus very important.
- The crossbar switches words, not bits. Each processing element acts on  $B$  bits, and the values switched by the crossbar are also  $B$  bits wide. Outside of using a PE resource, only the barrel-shifters can reorder bits within a word; they can shift a word to the left under configuration control. Thus, something like reversing the order of the bits in a word cannot be accomplished at all by using only one level of the interconnection network. (In this sense our interconnection network is more restricted than the networks of commercial FPGAs.)
- We have to map computations of arbitrary widths to a network and processing elements which only act on multiples of  $B$  bits each.

#### 4.3 The P&R Algorithm

The circuit entering P&R contains two types of operators: computation operators, which modify bits, and *routing-only* operators, which only reorder bits. Logical shift, bit selection and concatenation fall into the latter category. The goal of the P&R algorithm is to map all routing-only operators to the interconnection network. Only when this is not possible will routing-only operators consume processing resources. In the graph at the left of the Figure 11 we have shaded the computation operators; the white squares are routing-only operators.

P&R consists of several pre-processing steps followed by the actual placement. Here is how the pre-processing is done:

1. Early in the compilation process nodes that are too large to fit in a stripe (or whose carry chain is too long to meet the timing requirements) are broken into pieces, as indicated in Section 3.2.3.
2. The next phase transforms the graph to ensure that it is routable. The subgraphs comprised entirely of routing-only operators describe a permutation from source bits to

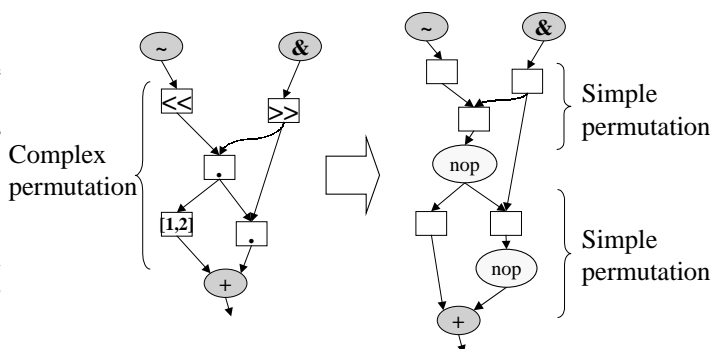


Figure 11: Inserting noops to break complex interconnection patterns into products of simpler ones.

destination bits<sup>3</sup>. If the transformation *could possibly* be too complex for the interconnection network to handle, then lazy noop operators are inserted into the graph. This transforms the permutation into a series of simpler permutations. This pass is implemented using pattern matching: it looks for complex patterns and splits them, as shown in Figure 11.

3. An *interference analysis* is performed on the graph. The goal of this pass is to ensure that all the source wires of a node are available at the crossbar. Interference analysis is a global pass, resulting in what is essentially lookahead capabilities for the greedy placement algorithm. On PipeRench the sources will be at the crossbar when the PEs that are sources of the wires are in different columns. The pass annotates the graph so that the placer will attempt to place nodes with a common descendant in separate columns.
4. Additional constraints are computed for placement scheduling order (e.g., place node  $a$  before node  $b$ ) and position (e.g., both inputs of a MUX cannot be placed in the rightmost column because the MUX needs one column more to the right to route the selection wire).

After these preliminary steps are carried out, the actual P&R is done:

5. Using list scheduling on the topologically sorted graph, each node which is a computation-operator is considered in turn for placement in the current stripe. When no node can be added in the current stripe, processing moves to the next one.
6. Before a node is placed, the compiler determines if it would create a combinational delay that exceeds the maximum allowed value. If so, the node is considered for placement in the next stripe, slicing the delay path with a pipeline register.
7. Before any placement of a node is attempted, the feasibility of placing a node is analyzed. This analysis is based on the position of the sources, which have already been placed.

<sup>3</sup>Strictly speaking it is a many-to-one function, because one bit value may be broadcast to several destinations, for instance when sign-extending a value.

This step asks the question: “could this node be placed in a completely empty stripe, assuming all the computation resources and the interconnection network are free?” This amounts to a test of the complexity of the interconnection pattern between the sources of the node and the node itself. For instance, a node whose input is obtained by the bit-reversal of the output of another node will never be placeable on PipeRench.

8. If the node is unplaceable, we look for lazy noops in the subgraph connecting the node to its sources. The pre-processing steps ensure that such a noop exists. All noops between the unplaceable node and its source are considered in the order of increasing distance from the unplaceable node. The first one that can be placed is selected, and turned into a “real” noop, which will consume PEs. This noop is next inserted into the list schedule, and the placement continues.

When this noop is placed in a PE, the effect is to decompose the permutation between the sources and the destination into two permutations, which are implemented using two interconnection networks: one between the sources and the noop and one between the noop and the node itself. This is a form of “divide et impera”: divide the permutation into simpler ones and place each of them.

A noop may be also used to move a value from one column of the pass-register bus to another one; this can relieve the contention for the input ports of the crossbar.

This entire algorithm works in a greedy fashion: a placed node is never moved or removed. The preprocessing steps and the noop insertions guarantee that a solution always exists (i.e. since all the permutation transformations between noop are very simple and the operators are small enough, we know that the “forward progress” rule is true.)

In the worst case, such a strategy can give rise to solutions with very low utilization. Backtracking or local optimization strategies might improve the result. However, we have observed in practice that the quality of our results is good, as shown in Section 5. The fast compilation makes this a reasonable trade-off since we can rely on hardware virtualization to accommodate configurations of any size.

## 5 Evaluation

The DIL compiler currently produces configurations for PipeRench. We evaluate its performance, measuring both compilation speed and resulting configuration efficiency, on 11 benchmarks which include kernels from a wide spectrum of applications:

**ATR** implements the shapsum kernel of the Sandia algorithm for automatic target recognition [26]. This algorithm is used to find an instance of a template image in a larger image and to distinguish between images which contain different templates.

**Cordic** is a 12 stage implementation of the Honeywell Timing Benchmark for Cordic vector rotations [11]. Given a vector in rectangular coordinates and a rotation angle in degrees, the algorithm finds a close approximation to the resultant rotation.

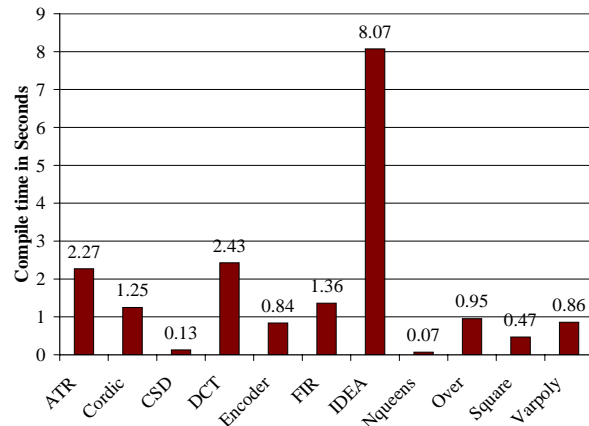


Figure 12: *Compilation time in seconds on PII/400Mhz.*

**CSD** implements a 16-bit canonical signed digit multiplier with the constant 123. CSD multiplication is an efficient replacement for traditional partial product multiplication in reconfigurable logic.

**DCT** is a one-dimensional, 8-point discrete cosine transform. The 2-D DCT is an important algorithm in Digital Signal Processing and is the core of JPEG image compression.

**FIR** is a FIR filter, like the one exhibited in Figure 4, but with 20 taps and 8-bit coefficients.

**IDEA** is the heart of Phil Zimmerman’s Pretty Good Privacy (PGP) data encryption. IDEA implements a complete 8 round International Data Encryption Algorithm with the key compiled into the configuration [24].

**Nqueens** is an evaluator for the n-queens problem on an 8x8 board. Given the coordinates of chess queens on a chessboard, it determines whether any of the queens are in line of sight to each other.

**Over** implements the Porter-Duff over operator [5]. It is a method of merging two images based on a mask of transparency values for each pixel.

**Square** simply squares a 16-bit signed number.

**Varpoly** evaluates a polynomial of degree three in  $x$ . The coefficients and  $x$  are supplied.

The target of our compilation is a PipeRench chip with  $B = 8$ -bit PEs, stripes 128 bits wide ( $N = 16$ ), a pass-register bus of  $8 * 16 * 8$  bits, and 31 stripes. The target critical path of PipeRench is set for 10ns (a clock of 100Mhz).

Figure 12 gives the total compilation times (from source to assembly) as measured on a PII/400 running Linux. Of upmost interest is the fact that the longest time (for the biggest kernel) is approximately 8 seconds, justifying our claim of fast compilation.

In Figure 13 we plot two magnitudes which have different units of measure. The dark bar is the speed measured in bit-operations per second, while the light bar is the size of the corresponding circuit in bit-operations. As mentioned, the



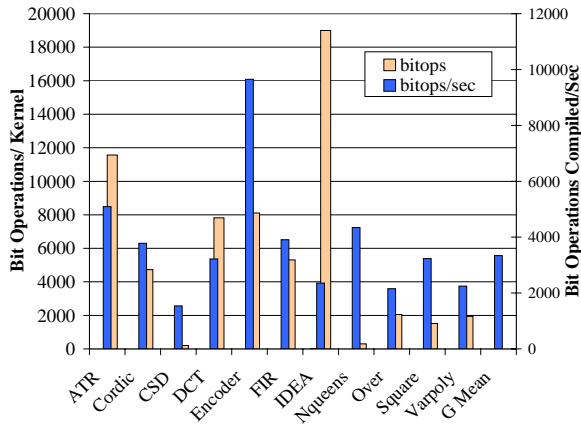


Figure 13: Compilation speed (PII/400Mhz) and circuit size. The geometric mean of the compilation speed is a little above 3000 bit-operations per second.

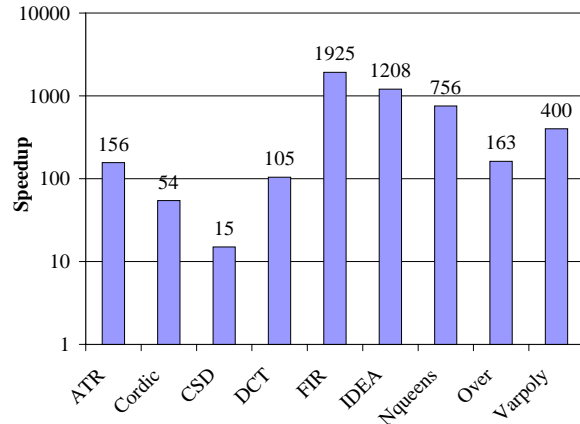


Figure 16: Speed-up of the application over a 300Mhz UltraSparc.

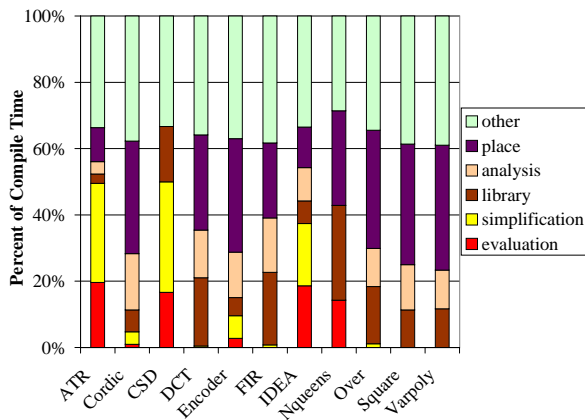


Figure 14: Compilation time breakdown for dominant phases.

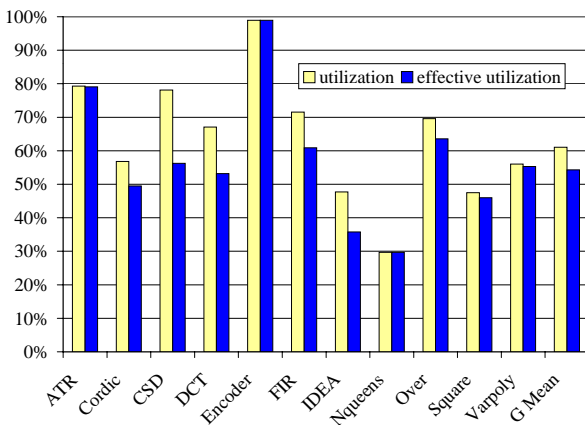


Figure 15: Hardware utilization of the generated code. The effective utilization discounts noops from taking useful space.

geometric average is above 3000 bit-operations/second. The encoder features an impressive speed because the DIL program itself has been generated automatically. The program therefore is very regular, is already optimized, and can be placed in a straightforward manner.

The Figure 14 shows how the compilation time is distributed among the most important phases of the compilation process. “Evaluation” includes parsing the source text, inlining the modules, unrolling the loops, and generating the graph. “Simplification” is the first round of constant-folding, type inference, width computation, etc. “Library” is the time taken to synthesize the operators from the library, including the subsequent simplifications. The “analysis” phase is the global analysis, which inserts the noops, computes the placement hints, and ensures the routability. “Place” is the actual placement procedure. “Other” is time taken by other general optimizations. We observe that the cumulative time (i.e. analysis+place) of the complete P&R algorithm is never much more than half of the total compilation time.

In Figure 15 we display the amount of hardware used by each application (i.e. how many of the PEs in each stripe are consumed). We have two bars: one for the utilized hardware, which averages a little more than 60% across all applications, and one for “effective utilization”. In measuring effective utilization we do not count the PEs used by noops as utilized (this is underestimating the utilization since some of the noops may actually be absolutely necessary because of the limited capabilities of the interconnection network). The effective utilization is approximately 55%, reflecting the fact that noops, on average, consume few resources. Notice also the excellent utilization for the ATR, CSD, Encoder, and FIR. In fact, Encoder and FIR are impossible to pack tighter, even by hand.

Figure 16 shows the simulated speed-up that the compiled configurations obtain when running on PipeRench over their equivalent C programs on a 300Mhz UltraSparc machine. The numbers are impressive. The bigger the benchmark, the more the parallelism is exploited, so the greater the speed-up. From the data, we conclude that virtualized pipelined architectures provide an enormous potential for performance improvements, and that we can compile to them.

Finally, we compare commercial tools (Synopsis's Design Analyzer combined with Xilinx's Design Manager targeting a Xilinx 4085XL) with DIL targeting PipeRench. To some extent we are comparing apples to oranges because the commercial tools are much more general. We do the comparison anyway to ensure that for our domain, the generation of pipelined datapaths, our approach is feasible. The DIL compiler creates the configuration for DCT 778 times faster than the commercial tools. Furthermore, the resulting configuration runs 10x faster on PipeRench than on the 4085XL.

## 6 Related Work

Many languages and systems have been proposed for programming RC systems. Most of these systems do not optimize for pipelined datapath creation and thus compile down to gates and then rebuild the datapaths from the gates. This often requires orders of magnitude more time than the DIL compiler takes to compile. Early work required this since it relied on vendor supplied tools for mapping, placement, and routing [13, 21, 2, 1, 16].

More recently, tools like Gama [6] and Napa-C [15] compile from C directly to the fabric. The DIL compiler differs from Gama in that it does not need to split the DAG into trees and can place more operators in a single stripe. Napa-C relies on pre-placed and routed modules and cannot optimize across these boundaries. The RaPiD-C compiler [9] also targets pipelineable datapaths, but it is oriented towards a much coarser-grained architecture. Unlike DIL, which uses linear time deterministic algorithms, the CORBA-ABS system uses simulated annealing [10] which often produces better results but is substantially slower. Other recent fast compilation systems, notably [12, 25, 27], are less focused on datapaths than on compiling traditional FPGA designs.

Among the many languages proposed to compile reconfigurable computing systems, DIL is most similar in spirit to Silage [17]. Unlike Silage, DIL can be used to express parameterized modules which the compiler can load on demand. DIL's library loading system is similar to recent module generators designs [8, 20, 4]. However, the DIL modules are written in the same language as the main program and remain relatively architecture-independent.

## 7 Future Work and Conclusions

In this paper we have presented the design and implementation of a compiler for a high-level language (DIL) whose targets are reconfigurable devices. DIL is expressive, extensible, and completely hardware independent. DIL is intended to be an intermediate step in the compilation from a high-level language like C.

We have shown that a fast linear time algorithm for place and route can yield good utilization, at least for the class of pipelined reconfigurable architectures. The compilation speed of approximately 3,000 bit-operations per second is orders of magnitude faster than traditional CAD tools. Our algorithm depends on two key features. First, it uses the concept of a lazy noop to ensure routability without requiring that the noop actually be inserted in the final configuration unless necessary. Second, global analysis creates annotations which guide the place and route process. In addition to a fast, effective place and route, the compiler implements a wide range of optimizations.

In the future we intend to extend the effectiveness of the compiler by performing time-multiplexing of the routing resources (similar to [3]), dealing with graphs with non-trivial cycles, and increasing the quality of the place and route algorithm, in particular by reducing register usage with graph-coloring based algorithms. We have also begun work on using DIL as an intermediate language in a compiler for C. Finally, we intend to test the generality of our algorithms by retargeting DIL to more traditional FPGA architectures.

## Acknowledgments

We would like to thank the members of the reconfigurable computing laboratory at CMU: Srihari Cadambi, Ronald Laufer, Matt Moe, Herman Schmit, Brian Showers, and Reed Taylor for their comments on earlier version of this paper and for generating kernels and simulation infrastructure. We offer a special thanks to Srihari for his contribution to the compiler. This work is supported in part by a grant from Altera Corporation and DARPA contract No. DABT63-96-C-083.

## References

- [1] J. M. Arnold. The Splash 2 software environment. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, Napa, CA, April 1993.
- [2] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
- [3] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Napa, CA, April 1993.
- [4] P. Bellows and B. Hutchings. JHDL—an HDL for reconfigurable systems. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175–184, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [5] Jim Blinn. Fugue for MMX. *IEEE Computer Graphics and Applications*, pages 88–93, March-April 1997.
- [6] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 123–132, Feb 1998.
- [7] D. Cherepacha and D. Lewis. A datapath oriented architecture for FPGAs. In *Second International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1994.
- [8] M. Chu, M. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object oriented circuit-generations in Java. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 158–166, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [9] D.C. Cronquist, P. Franklin, S.G. Berg, and C. Ebling. Specifying and compiling applications for RaPiD. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 116–127, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [10] A.A. Duncan, D.C. Hendry, and P. Gray. An overview of the CORBA-ABS high level synthesis system for multi-FPGA systems. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE*

- Workshop on FPGAs for Custom Computing Machines*, pages 106–115, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [11] Sanjaya Kumar et.al. Timing sensitivity stressmark. Technical Report CDRL A001, Honeywell, Inc., January 1997. <http://www.htc@honeywell.com/projects/acsbench/>.
- [12] Stephan W. Gehring and Stefan H.-M.-Ludwig. Fast integrated tools for circuit design with FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 133–139, Feb 1998.
- [13] M. Gokhale and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 399–408, Oxford, England, August 1995. Springer.
- [14] Maya Gokhale. High level compilation for fine grained FPGAs. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 165–173, April 1997.
- [15] M.B. Gokhale and B. J.M. Stone. NAPA-C: Compiling for a hybrid RISC/FPGA architecture. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 126–137, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [16] S. A. Guccione and M. J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, Napa, CA, April 1993.
- [17] Paul N. Hilfinger. Silage: A language for signal processing, 1985.
- [18] T. Isshiki and W.W.-M. Dai. Bit-serial pipeline synthesis for multi-FPGA systems with C++. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 38–47, April 1996.
- [19] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
- [20] O. Mencer, M. Morf, and M.J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 167–174, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [21] I. Page and W. Luk. Compiling occam into FPGAs. In *FP-GAs. International Workshop on Field Programmable Logic and Applications*, pages 271–283, Oxford, UK, September 1991.
- [22] B. Pottier and J.-L. Llopis. Revisiting Smalltalk-80 blocks: a logic generator for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 48–57, April 1996.
- [23] Herman Schmit. Incremental reconfiguration for pipelined applications. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 47–55, Napa, CA, April 1997.
- [24] Bruce Schneier. The idea encryption algorithm. *Dr. Dobbs's Journal*, 18(13):50, 52, 54, 56, December 1993.
- [25] J.S. Swartz, V. Betz, and J. Rose. A fast routability-driven router for FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 140–149, Feb 1998.
- [26] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.
- [27] R. Woods, S. Ludwig, J. Heron, D. Trainor, and S. Gehring. FPGA synthesis on the xc6200 using iris and trianus/hades (or from heaven to hell and back again). In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–164, Napa, CA, April 1997.