Efficient Place and Route for Pipeline Reconfigurable Architectures

Srihari Cadambi*and Seth Copen Goldstein†
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213.

Abstract

In this paper, we present a fast and efficient compilation methodology for pipeline reconfigurable architectures. Our compiler back-end is much faster than conventional CAD tools, and fairly efficient. We represent pipeline reconfigurable architectures by a generalized VLIW-like model. The complex architectural constraints are effectively expressed in terms of a single graph parameter: the routing path length (RPL). Compiling to our model using RPL, we demonstrate fast compilation times and show speedups of between 10x and 200x on a pipeline reconfigurable architecture when compared to an UltraSparc-II.

1. Introduction

Current workloads for computing devices are rapidly changing to include real-time media processing and compute-intensive programs. These new workloads are dataflow-dominated and characterized by regular computations on large sets of small data elements, limited I/O and lots of computation. As a result, many of them seriously underutilize the large datapaths and instruction bandwidths of conventional processors.

These problems are being addressed through processor changes [12, 6, 18]. However, reconfigurable computing offers a fundamentally different way to solve these issues. Reconfigurable architectures have the capability to configure connections between programmable logic elements, registers and memory in order to construct a highly-parallel implementation of the processing kernel at run-time. This feature makes them very attractive, since a specific high-speed circuit for a given instance of an application can be generated at compile- or even run-time.

In this paper, we present a generalized compilation scheme for a class of reconfigurable architectures that exhibit *pipeline reconfiguration*. The target pipeline reconfigurable architecture is represented by a simple VLIW-style model with a list of architectural constraints. We heuristically represent all constraints by a single graph parameter, the *routing path length* (RPL) of the graph, which we then attempt to minimize. Our compilation scheme is a combination of high-level synthesis heuristics and a fast, deterministic place-and-route algorithm made possible by a compiler-friendly architecture. The scheme may be used to design a compiler for any pipeline reconfigurable architecture, as well as any architecture that may be represented by our model. It is very fast and fairly efficient.

The remainder of the paper is organized as follows: Section 2 describes the concept of pipeline reconfiguration and its benefits. We briefly describe PipeRench, an instance of pipeline reconfigurable architectures, and the VLIW-style architectural model for the compiler. Section 3 describes the compiler along with our place-and-route algorithm. Section 4 presents results that establish correlation between the RPL and our objective function. We also show the effects of different priority function parameters on our results, and report speedup numbers over an UltraSparc-II obtained by using our compiler on the PipeRench reconfigurable architecture. Section 5 discusses some related work and we conclude in Section 6.

2. Pipeline Reconfigurable Architectures

In this section, we review pipeline reconfiguration [14], a technique in which a large physical design is implemented on smaller sized hardware through rapid reconfiguration. This technique allows the compiler to target an unbounded amount of hardware. In addition, the performance of a design improves in proportion to the amount of hardware allocated to that design. Thus, as more transistors available, the same hardware designs achieve higher levels of performance.

*cadambi@ece.cmu.edu
†seth@cs.cmu.edu

0-7695-0801-4/00 $10.00 © 2000 IEEE

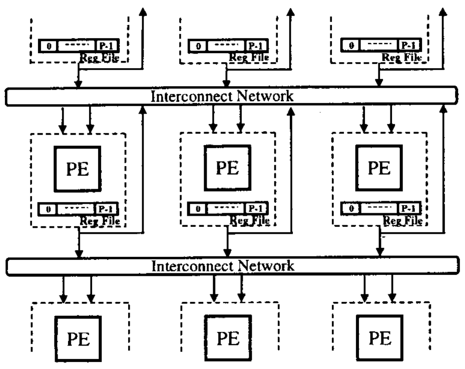

Figure 1. PipeRench architecture showing the N PEs in each stripe and their register files. The interconnect switches B-bit values. Unless overwritten by their PE, the register files constitute a pipelined bus.

Pipeline reconfiguration is a method of virtualizing pipelined hardware applications by breaking the configuration into pieces that correspond to pipeline stages in the application. These configurations are then loaded, one per cycle, onto the interconnected network of programmable processing elements (PEs) collectively known as the *fabric* [5].

Since the configuration of stages happens concurrently with the execution of other stages, there is no loss in performance due to reconfiguration. As the pipeline is filling with data, stages of the computation are configured one step ahead of the data [5].

We now present the main characteristics of PipeRench, an instance of the class of pipeline reconfigurable architectures. We then present our architectural model that describes this class of architectures for the compiler.

2.1. PipeRench: an instance of Pipeline Reconfigurable Architectures

PipeRench is composed of pipeline stages called *stripes.* The pipeline stages available in the hardware are referred to as *physical stripes* while the pipeline stages that the application is compiled to are referred to as *virtual stripes.* The compiler-generated virtual stripes are mapped to the physical stripes at run-time. Each physical stripe is composed of N processing elements (PEs). In turn, each PE is composed of B identically configured look-up tables (LUTs), P B-bit registers in a register file, and some control logic. The PEs have 2 data inputs. 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 register-file interconnect,* allows the values of all the registers to be transferred to the registers of the PE in the same column of the next stripe. Figure 1 shows the PEs, their register files and the interconnection network in PipeRench.

2.2. VLIW-style Architectural Model

Despite hardware virtualization, three important constraints are still imposed by such a pipeline reconfigurable architecture on the compiler: (i) a limited number of PEs per stripe, (ii) a limited number of registers per PE and (iii) a limited number of read and write ports in each register file. These restrict the number of values that may overlap (i.e., live ranges of variables) on the register file of a single PE, make routing difficult, and impose a constraint on operator placement.

We can model this as a distributed register file VLIW-like architecture with N B-bit wide functional units (FUs, or PEs in our case). The configuration for each stripe is analogous to a VLIW-instruction. Each instruction is comprised of N sub-instructions, one for each FU. A sub-instruction determines the operation of the FU, the sources of its inputs, and the register allocated for its outputs. Each FU has one register file with P registers, R read ports and W write ports. The inputs to a FU may come from other functional units depending on the interconnect. Thus, by generalizing this model and varying the parameters N, B, P, R and W, our work may be extended to compilers that map dataflow graphs to other similar architectures.

2.3. Definitions

Before we present the problem and our solution, we define a few terms. A dataflow graph is used to represent the input program. It consists of nodes which represent operations[1]. Nodes are interconnected by directed edges, referred to as *wires.* A wire has a single source node but can have multiple destination nodes *(fanout).* Each wire carries a value, represented by multiple bits.

Nodes may be *routing nodes* or *non-routing nodes.* A routing node does not consume any functional resources, while a non-routing node has to be allocated PEs. An example of a routing-node is a "bit-select" (marked [1,1] in Figure 2).

The *routing path length* of a single bit in a given wire w is the total number of time-steps between w and its nearest non-routing source node. The *routing path*

[1] We use the the terms "Node" and "Operation" interchangeably in this paper.

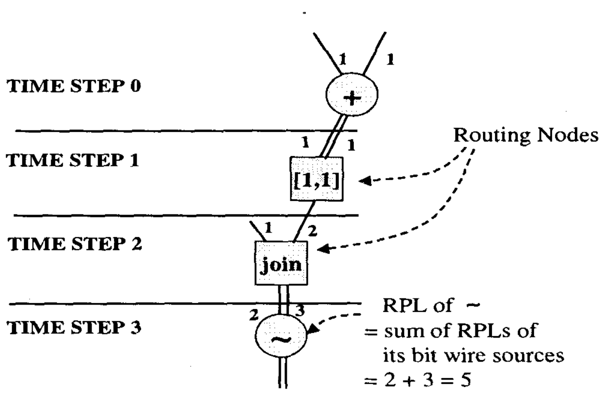

Figure 2. Illustration of RPL. The right source bitwire of the ~ operator has its nearest non-routing source node in time-step 0, resulting in a RPL of 3. The left source bit of the ~ time-step 1, resulting in an RPL of 2. Each arc represents one bit of a wire.

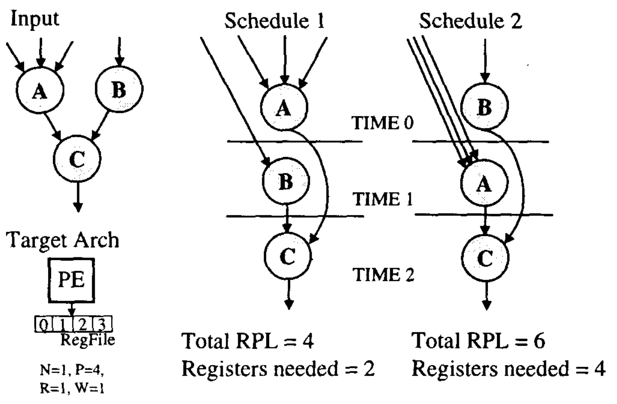

Figure 3. Effect of minimizing routing path length (RPL) on the maximum number of registers needed. The target architecture has 1 PE per stripe. Schedule 1 prefers to place the high fan-in node A first, while schedule 2 postpones A. The total RPL and maximum number of registers in Schedule 1 is lower.

length of a node is the sum of the routing path lengths of all of the bits of its source wires. (See Figure 2.)

2.4. The Problem and Solution

The problem that we solve is to map a given dataflow graph to a pipeline reconfigurable architecture defined by the model in Section 2.2. Specifically, for each operation we have to first assign a stripe. Within that stripe, we have to allocate PE resources to perform the operation, interconnect resources to route its inputs and outputs and a register from the register file for storing its live outputs. The objective is twofold: (i) minimize the number of virtual stripes while adhering to the resource constraints and (ii) obtain fast compilation speeds.

Our solution is based on minimizing RPL. We qualitatively explain the effect of RPL on our objective function, and substantiate this empirically in Section 4. The RPL of a node is related to the lifetime of its input wires. As long as a wire is live, it is allocated a register on a pipeline reconfigurable architecture. Minimizing the overall RPL of a netlist thus leads to lower register pressure. Further, if a new virtual stripe S is added to the schedule, there is at least one extra wire crossing the boundary of S, leading to an increase in the overall RPL. Thus, lowering the RPL not only has an effect on lowering the number of virtual stripes, but also tends to reduce the number of wires crossing the boundary between any two virtual stripes, reducing interconnect demand. A simple example of the effect of RPL on register usage is shown in Figure 3. We now present the following solution:

- We take advantage of hardware virtualization and assume an infinite amount of hardware in one dimension, that is, unlimited stripes. This allows us to gainfully adopt a greedy, deterministic, linear time place-and-route algorithm in order to get a lot of speed. The algorithm never backtracks or removes an operation that was already placed.
- In order to adhere to the constraints and minimize the number of virtual stripes while still maintaining the speed of the algorithm, we determine an order for the operations to be placed which aims at minimizing the overall RPL.

We thus adopt a three-step strategy for place-and-route (See Figure 4). The algorithm takes as input a topologically sorted dataflow graph. The first step, graph preprocessing, annotates each operator with possible placement locations (See Section 3.2.1). In the second step, the algorithm uses list scheduling[9] based on RPL. The nodes selected are placed on the architecture while satisfying functional, register and routability resource constraints. Nodes which cannot be placed owing to routability constraints are handled in the third step. The output is a dataflow graph that is placed on the pipeline reconfigurable architecture. Using this method, a compiler back-end for any pipeline reconfigurable architecture may be designed.

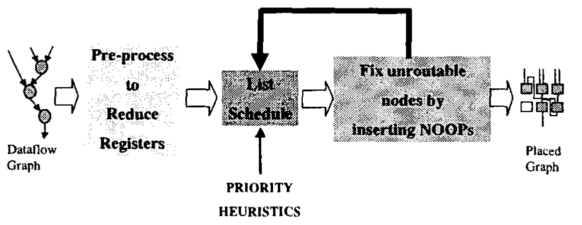
Dataflow Graph
Pre-process to Reduce Registers
List Schedule
Fix unroutable nodes by inserting NOOPs
Placed Graph
PRIORITY HEURISTICS

Figure 4. Place and route for pipeline reconfigurable architectures.

3. The Place-and-Route Phase and the Compiler

We describe our place-and-route algorithm in terms of the DIL compiler for PipeRench.

3.1. The DIL Compiler

The DIL compiler compiles a high-level, architecture independent Dataflow Intermediate Language (DIL) and produces configurations. DIL targets pipeline reconfigurable architectures. It is intended to be used both by programmers and as an intermediate language for a high-level language compiler.

The compiler first reads in architectural details like the number of PEs, bit-widths of each PE, number of registers, as well as the target clock-cycle. It then reads in an input design described in DIL and converts it into a dataflow graph consisting of nodes and wires.

The main stages of the compiler are shown in Figure 5. Each operation in the dataflow graph is synthesized in terms of canonical operations that may be mapped to a processing element on the target architecture. For instance, if the target architecture is fine-grained and has only ALUs, multiplications are decomposed into a series of shifts and adds. Following this, the graph is subjected to a decomposition pass where operations too wide to fit in a clock-cycle are broken up and pipelined into smaller operations. It then goes through a series of optimizations such as common subexpression elimination and constant folding. The rest of this paper focuses on the main part of the back-end, place-and-route. Details of the DIL language and the compiler front-end may be found in [2].

3.2. Place and Route

As mentioned in Section 2.4 and depicted in Figure 4, we propose a graph pre-processing step followed by a placement scheme based on list-scheduling.

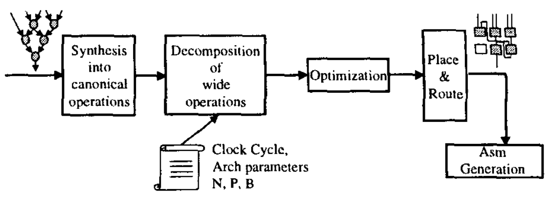
Synthesis into canonical operations
Decomposition of wide operations
Optimization
Place & Route
Asm Generation
Clock Cycle, Arch parameters N, P, B

Figure 5. The main stages of the DIL compiler.

3.2.1 Pre-processing

This pass imposes constraints on nodes and reduces placement possibilities. These constraints are placement directives which heuristically tag certain placement locations on the chip as unfavorable. They are determined in advance by examining the architectural constraints. Figure 6 shows one example of these constraints for PipeRench.

3.2.2 The RPL-based Priority Function

Prioritizing the nodes well is very important since the subsequent place-and-route algorithm does not backtrack. We propose a heuristic that attempts to minimize the RPL at every step in the place-and-route process. First, the following four sets of nodes are identified:

- The set of scheduled nodes, $\mathbf{S} = \{$n: n is placed$\}$
- The set of ready nodes, $\mathbf{R} = \{$n: all predecessors of $n \in S\}$
- The set of "almost" ready nodes, $\mathbf{A} = \{$n: exactly one predecessor of $n \in R$, all other predecessors $\in S\}$
- The set of nodes with at least one predecessor in S or R, $\mathbf{B} = \{$n: all predecessors of $n \in \{SUR\}\}$

Note that $A \subseteq B$ and that $S \cap R \cap B = \emptyset$. Results with each of these RPL-based priority functions are given in the next section.

At each step of the placement algorithm, we consider the nodes in R for allocation in a partially occupied stripe P. For each node n in R, the cost C is based on the estimated increase in the RPL if n were not placed in P. We define three methods to compute the cost:

- No Prediction: C = increase in the RPL of n alone
- Single-Level Prediction: C = increase in the RPL of n and all children m of n such that $m \in A$.
- Two-level Prediction: C = increase in the RPL of n and all children m of n such that $m \in B$.

3.2.3 Handling Unroutable Nodes

Owing to our greedy approach, occasional situations do arise when the inputs of a ready node cannot be routed to, and no further ready nodes are available to

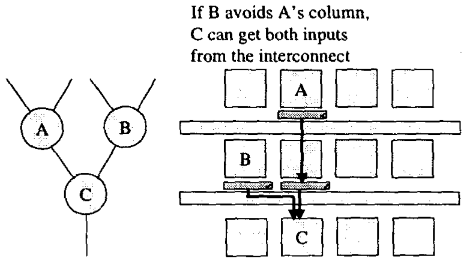
If B avoids A's column, C can get both inputs from the interconnect
A
B
C
A
B
C

Figure 6. If B avoids the column in which A is placed, C can get its inputs on the interconnect since the inputs are in distinct register files, each with a single read port.

place. Under such circumstances, a NOOP is inserted at one input in order to ensure routability [2].

Place-and-route results with RPL as well as the final speedups obtained by running our compiled applications on PipeRench are presented in Section 4.

4. Results

In this section, we present the results of our compiler design for pipeline reconfigurable architectures. There are three parts to our results. First we empirically demonstrate a close correlation between the overall RPL of a circuit and the number of virtual stripes it is compiled to. Next, we show improvements in the number of virtual stripes using our RPL-based method. Finally, speed-ups seen over a 300 MHz SparcII microprocessor are presented.

All the data presented here was gathered for implementations of various kernels on the PipeRench architecture. The kernels were chosen based on demand for the applications in the present and near future, their recognition as industry performance benchmarks, and their ability to fit into our computational model.

ATR implements the shapesum kernel of the Sandia algorithm for automatic target recognition [17]. This algorithm is used to find an instance of a template image in a larger image, and to distinguish between images that contain different templates.

Cordic is a 12 stage implementation of the Honeywell timing benchmark for Cordic vector rotations [7]. Given a vector in rectangular coordinates and a rotation angle in degrees, the algorithm finds a close approximation to the resultant rotation.

DCT is a one-dimensional, eight-point discrete cosine transform [8]. **DCT-2D**, a two-dimensional DCT, is an important algorithm in digital signal processing and is the core of JPEG image compression.

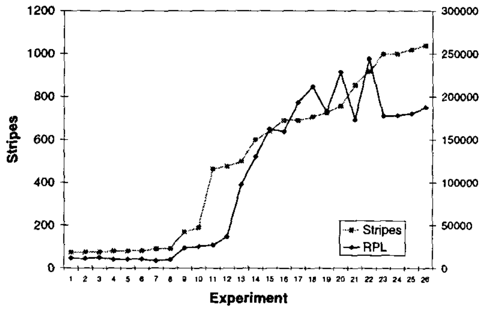

Figure 7. Correlation between RPL and stripes.

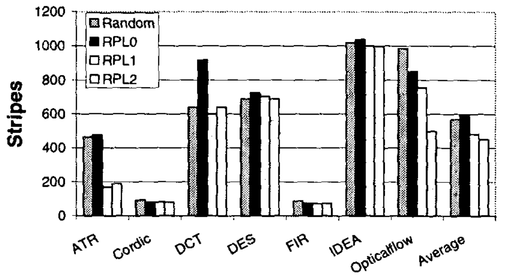

Figure 8. Place and route improvements seen with the RPL-based method.

FIR implements a FIR filter with 20 taps and 8-bit coefficients.

IDEA implements a complete eight-round International Data Encryption Algorithm with the key compiled into the configuration [15]. IDEA is the heart of Phil Zimmerman's Pretty Good Privacy (PGP) data encryption.

Over implements the Porter-Duff over operator [1]. This is a method of joining two images based on a mask of transparency values for each pixel.

PopCount return the number of 1's in a binary word.

Correlation between RPL and Virtual Stripes We conducted several experiments with the above benchmarks in order to establish the relationship between RPL and virtual stripes. The results are shown in Figure 7.

Improvements seen with RPL Here we present improvements seen with the compiler having a priority function based on RPL-minimization (Section 3.2.2).

These results are shown in Figure 8. For each application, there are 4 bars, showing the number of stripes with a random priority heuristic, RPL-based heuristic with no prediction, single-level prediction and two-level prediction. It is seen that on the average, RPL-based heuristics with no prediction are worse than a random

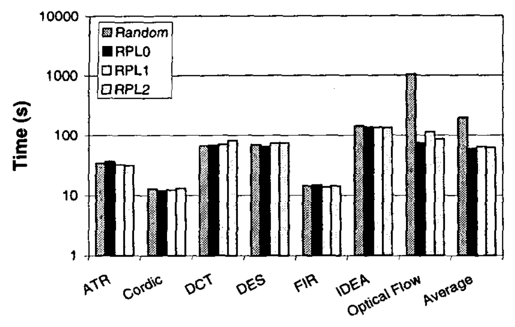

Figure 9. Compiler running times with RPL-based heuristic.

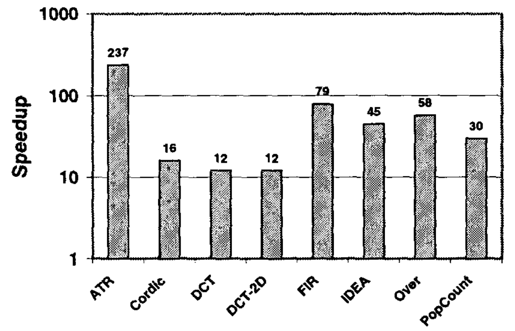

Figure 10. PipeRench's simulated performance vs. a 300 MHz UltraSparc-II. Raw speedups achieved using our compiler for a 100 MHz PipeRench, with parameters $\{N, B, P, W, R\} = \{16, 8, 8, 1, 1\}$.

heuristic. However, prediction improves their performance of RPL-based heuristics, with single and two-level predictions being 15.2% and 20.6% better than the random case respectively. Figure 9 shows the compiler running times with the random and RPL-based methods. It may be observed that on the average, the second-level prediction is actually a little *faster* than the other cases. This is because RPL-based prediction leads to better placement, and thus fewer failures. Fewer failures means that nodes have to be postponed fewer times, which decreases overall place-and-route time.

Speedups Obtained over a Microprocessor We compiled to a PipeRench architecture with 8-bits per PE, 8 PEs per stripe, 8 registers in the register file of each PE, a single write port from each PE to its register file and a single read port from each register file to a complete crossbar interconnect. The simulated performance of PipeRench based on a 100 MHz clock speed and the number of virtual stripes is compared to a Sun UltraSparc-II running at 300 MHz. Figure 10 shows the raw speedup for all kernels. Although the performance may not be same after considering memory and I/O bottlenecks, a large fraction of the raw speedup is achievable [5].

In addition, our place-and-route is also orders of magnitude faster than commercial CAD tools. For instance, we can compile the DCT in less than 10 seconds, while Xilinx Design Manager targeting a Xilinx 4K series FPGA takes over 1 hour for the same application.

5. Related Work

A wealth of related work exists in the realm of high-level synthesis for FPGA-based systems [11, 13]. [13] surveys various logic synthesis methods targeted at FPGA architectures. The emphasis there was on developing tools that minimize the combinational part of design, and not on pipeline optimization, performance or routability, which is the case with our work. Force-directed scheduling (FDS) [11] provides a methodology to schedule nodes in time-slots such that the resource usage in each time-slot is balanced. However, we found that it is difficult to formulate an FDS approach for our architecture, given the register-file and register port constraints.

[16] describes a fast router for island-style FPGAs while [10] describes a performance-driven simultaneous place-and-route methodology. The similarity here is that our place-and-route algorithm also performs simultaneous place-and-route: placement is completed only if routing is possible. [10] describes a set of new techniques for row-based and island-style FPGAs. The techniques rely on iterative improvement augmented with fast complete timing heuristics. Earlier work on PipeRench [2] describes the DIL language and front-end phases of the hardware compiler. [3] describes fast module mapping and placement for datapath slices in FPGAs, where the modules are placed simultaneously with the mapping.

[4] describes a hypergraph coloring algorithm to allocate variables to a distributed register-file VLIW architecture. Our method accomplishes much the same thing using RPL, but is faster.

6. Conclusions

In this paper, we present a hardware compiler for pipeline reconfigurable architectures. Such architectures are compiler-friendly and provide an infinite amount of hardware in one dimension. However, they still have constraints like the width of the pipeline stage, the interconnect, limited registers and limited register file ports.

We present a compilation scheme based on three-steps: graph pre-processing, prioritizing of the nodes followed by a greedy, linear, deterministic place-and-route. We also present a VLIW-like model that captures the architectural constraints inherent in such architectures, effectively describing the architecture to the compiler. Instead of attempting to solve for all the architectural constraints, we formulate an approach to simply minimize the overall routing path length (RPL) in the graph that represents the netlist. The RPL formulation effectively captures architectural constraints like limited registers and limited register ports. Minimizing the overall RPL tends to lower the number of virtual stripes quickly, which is the final objective.

With our compiler targeting a 100 MHz PipeRench reconfigurable architecture, we measure and compare the estimated performance against a Sun UltraSparc-II microprocessor running at 300 MHz. We obtain impressive speedups across a suite of representative kernels, with very fast compilation speeds.

Acknowledgements

The authors would like to thank the members of the CMU PipeRench group for their contributions to this work. This research is supported by DARPA contract DABT63-96-C-0083.

References

[1] Jim Blinn. Fugue for MMX. *IEEE Computer Graphics and Applications*, pages 88–93, March-April 1997.

[2] M. Budiu and S.C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA '99)*, Monterey, CA, Feb. 1999.

[3] 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.

[4] Andrea Capitano, Nikil Dutt, and Alexandru Nicolau. Partitioning of variables for multiple-register-file vliw architectures. In *International Conference on Parallel Computing*, pages 298–301, 1994.

[5] Seth C. Goldstein, Herman Schmit, Matt Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *The 26th Annual Internation Symposium on Computer Architecture*, pages 28–39, May 1999.

[6] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[7] Sanjaya Kumar and et al. Timimg sensitivity stressmark. Technical Report CDRL A001, Honeywell, Inc., January 1997. `http://www.htc.honeywell.com/projects/acsbench/`.

[8] C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Proc. International Conference on Acoustics Speech, and Signal Processing 1989 (ICASSP '89)*, pages 9880–991, 1989.

[9] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc, 1994.

[10] Sudip K. Nag and Rob A. Rutenbar. Performance-driven simultaneous place and route for fpgas. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 6*, pages 499–518, June 1998.

[11] Pierre G. Paulin and John P. Knight. Force-directed scheduling for behavioral synthesis of asics. In *IEEE Transactions on Computer-Aided Design, Vol. 8, No. 9*, June 1989.

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

[13] Alberto Sangiovanno-Vincentelli and Jonathan Rose. Synthesis methods for field-programmable gate arrays. In *Proceedings of the IEEE, Vol. 81, No. 7*, pages 1057–83, July 1993.

[14] Herman Schmit. Incremental reconfiguration for pipelined applications. In *Proceedings of the IEEE Symposium for FPGAs for Custom Computing Machines*, pages 47–55, April 1997.

[15] Bruce Schneier. The IDEA encryption algorithm. *Dr. Dobb's Journal*, 18(13):50, 52, 54, 56, December 1993.

[16] 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.

[17] 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.

[18] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, March 1996.