Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs

Seda Ogrenci Memik
Computer Science Department
UCLA
seda@cs.ucla.edu

Gokhan Memik
Dept. of Electrical Engineering
UCLA
memik@ee.ucla.edu

Roozbeh Jafari, Eren Kursun
Computer Science Department
UCLA
rjafari@cs.ucla.edu, kursun@cs.ucla.edu

ABSTRACT

In this paper we discuss the global resource sharing problem during synthesis of control data flow graphs for FPGAs. We first define the Global Resource Sharing (GRS) problem. Then, we introduce the Global Inter Basic Block Resource Sharing (GIBBS) technique to solve the GRS problem. We developed five heuristics to solve the GRS problem. The first tries to minimize the number of connections between modules, the second considers the area gain, the third uses the criticality of operations assigned to resources as a measure for deciding on merging any given pair of resources, the fourth tries to capture common resource chains and overlap those to minimize both area and delay, and the fifth is the combination of these heuristics. While applying resource sharing, we also consider the execution frequency of the basic blocks. Using our techniques we synthesized several CDFGs representing applications from MediaBench suite. Our results show that, we can reduce the total area requirement by 44% on average (up to 59%) while increasing the execution time by 6% on average.

Categories and Subject Descriptors

B. [**Hardware**]: Register-Transfer-Level Implementation

General Terms

Algorithms,Design

Keywords

Resource Sharing, FPGA, Control Data Flow Graph

1. INTRODUCTION

In this paper, we present Global Inter Basic Block Resource Sharing (GIBBS), a method for global resource sharing during automatic synthesis of control data flow graphs (CDFGs). This method is integrated within the automatic synthesis of high-level application descriptions targeting programmable hardware.

Over the past decade, Field Programmable Gate Arrays (FPGAs) have evolved at a racing pace. Designers are now able to use these devices to implement complex circuitry. Traditionally, designs were mapped onto FPGAs manually. However, with changes in the nature of designs and the increasing complexity of FPGA hardware, manual mapping is becoming cumbersome. As a result, increasing the level of abstraction for designers and automating the mapping process emerges as an attractive option. We believe that GIBBS will be an integral part of such automated synthesis tools for next generation FPGAs. In this paper, we propose a global resource sharing technique to complement an automatic hardware compilation flow with high-level synthesis integration. We particularly focus on the impact of high-level planning for resource sharing on the quality and feasibility of the final designs. The compiler stage in such flows performs several optimizations to leverage the transition from algorithmic descriptions to hardware. Nevertheless, many possible optimizations require knowledge from low-level tools. Performing resource sharing considering delay and interconnect is one such optimization. Therefore, we implement our optimizations within the high-level synthesis.

For many target applications, resource sharing must be performed in order to achieve feasible designs. Assuming infinite resources and mapping each computation to a dedicated module may be infeasible for devices with a fixed area such as FPGAs. On the other hand, applying aggressive resource sharing strategies may degrade the design quality, because resource sharing introduces multiplexer components, which can introduce additional delay on the critical path of execution. If resource sharing is too aggressive, these multiplexer may also increase the amount of interconnect. Naturally, ASIC technology can benefit from a well planned resource sharing strategy as well. Synthesis for FPGAs distinguishes from ASICs in the following aspect. *Possible savings in area and interconnect through resource sharing is an issue of manufacturing cost for synthesizing ASICs while there is always stringent design constraints. However, for FPGAs this is really an issue of feasibility with so much pre-characterization and pre-fabricated features existing in*

DAC 2003, June 2-3 2003, Anaheim, California USA

the target technology.

Specifically, our contributions in this paper are:

- We introduce the concept of inter basic block resource sharing during automatic synthesis of CDFGs targeting FPGAs.
- We provide heuristics for global inter basic block resource sharing. We incorporate profiling information into our optimization process.
- We present experimental results for representative applications measuring the effectiveness of the proposed techniques.

The rest of this paper is organized as follows. Section 2 gives an overview of existing automatic synthesis paths from high-level descriptions to programmable hardware. We also summarize some existing work on the problem of resource sharing in the global sense. We provide background information related to our work in Section 3. The problem of global resource sharing will be formally defined in Section 4. We will describe our algorithm for global resource sharing in Section 5. In Section 6 we present experimental results. Section 7 concludes the paper with a summary.

2. RELATED WORK

Early efforts in automatic mapping of computation onto programmable logic dealt with extracting customized instructions from an input program and assigning those to a programmable co-processor while the remainder of the program is compiled targeting a CPU. Computationally intensive kernels were considered for hardware implementation and it was the task of the compiler to identify those portions of the input code. Compilers accompanied by their respective novel processor architectures like PRISM [13], Garp [3], [2], NAPA C compiler [5], and Chimaera [14] were proposed.

Other projects proposed automatic datapath generation from high level descriptions such as Match [6], Cameron [7], and DEFACTO [10].

Resource sharing in the global context, i.e., with respect to a CDFG has not been incorporated into existing hardware compilers targeting programmable systems. There are some proposed techniques for efficient resource sharing in high-level synthesis of ASICs. Kim et al. [8], proposed a technique to transform a data flow graph with conditional branches into an equivalent representation without conditional branches. This transformation involves resource sharing between operations from mutually exclusive parts of conditional branches. The execution model of this technique treats the complete data flow graph with conditional branches as a flattened entity. As we will describe our execution model in Section 3, it will be clear that it is fundamentally different than their model. Kim et al. do not consider neither the input dependent behavior of the control flow nor the interconnect complexity of the resource sharing decisions in their technique. Raje and Bergamaschi [11] proposed an algorithm to perform resource sharing for both registers and functional units taking interconnect and multiplexer costs into account. Their execution model is similar to that of Kim et al. and again the fact that different execution paths can be executed with different frequencies is not considered. In their work, they aimed to combine interconnect and area within a single cost function, while we explored the benefits and shortcomings of different optimization objectives including, but not limited to interconnect and area within a variety of heuristics.

3. BACKGROUND

In a generic flow for automatic mapping of application onto programmable hardware, the application described in a high-level programming language is processed by the compiler. The compiler generates an *internal representation* (IR). While internal representations in different compilers take different forms and names, essentially they capture two basic pieces of information about an application: control flow and data dependency. A control data flow graph produced by the compiler stage provides this information to the high-level synthesis step. Next, high-level synthesis generates a *Register Transfer Level* (RTL) description of the design. Back-end tools perform logic synthesis and physical synthesis on this RTL description and create the bit-stream data to program the target device.

In this work on our focus is within the high-level synthesis stage. This stage contains major tasks related to generation of datapath and control logic based on the information provided by the compiler. The input to the high-level synthesis stage is a CDFG. Figure 1 (a) gives an example of a CDFG. This computation model contains both data and control dependencies within a computation. In our CDFG representation, each node corresponds to a basic block. The edges of the CDFG represent the control precedence between the basic blocks. In turn, each basic block node has an internal DFG representation. These DFGs capture the actual computation and the data dependencies within the application.

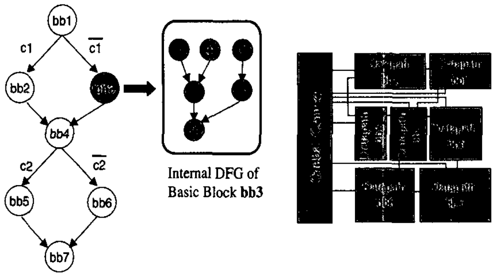
Our execution model is based on the sequential execution of basic blocks. Our synthesis methodology generates datapaths for each individual basic block while exploiting the parallelism within basic blocks. [1] When the execution proceeds from basic block bb_i to basic block bb_j, basic block bb_i is responsible for generating an *enable* signal that will initiate the execution of basic block bb_j. Each basic block starts execution when its enable signal is asserted. If a basic block is reachable through multiple possible execution paths, then enable signals from corresponding basic blocks preceding the basic block in control flow are *OR-ed* together to generate the enable signal. Finally, a global control unit is used for the remaining global control signals such as reset, initializations, etc.

There are certain practical reasons behind our choice of this particular execution model. First, we are attempting to map a significantly large and complex portion of an application (even there may be complete complete applications in some cases) onto hardware. Hence, the intermediate representation is bound to contain tens, even hundreds of basic blocks. Using a flat CDFG representation, scheduling, binding and synthesizing control efficiently both in terms of design quality and run-time is challenging.

Since our target hardware is an FPGA, logic and routing area is restricted. Under these circumstances, resource sharing is not an optimization option, instead a requirement. We need an efficient way of sharing functional units among datapaths of basic blocks. However, sharing should not introduce large critical path delays and a larger demand on intercon-

---

[1] In order to increase parallelism, entities containing multiple basic blocks (e.g., hyperblocks [9]) can be equivalently given to our high-level synthesis tool as input.



(a) (b)

Figure 1: (a) An example CDFG: each node is a basic block containing an internal DFG. (b) Illustration of our execution model.

nect and logic resources due to additional multiplexers to overshadow the anticipated benefit. Moreover, not every execution path is invoked in equal frequency. Resource sharing should consider the criticality of individual basic blocks. In addition, resource sharing may reduce the overall execution time by reducing the average interconnect length.

4. PROBLEM FORMULATION

In this section we present a formal definition of GRS. GRS refers to reusing a functional unit within datapaths of multiple basic blocks. We formulate the GRS problem as follows:

Input: a CDFG, a library of functional modules and multiplexers $M = \{m_i | module - delay(d_i), module - area(a_i)\}$*, and the execution frequency of each basic block.*

GSR is to find a set of modules to be shared among multiple basic blocks, such that the decrease in area requirement is maximized while the increase in the expected latency of the CDFG is less than Δ.

Λ, the increase in the expected latency of a CDFG is equal to $\Lambda = \sum_j f_j \times \delta_j$, where f_j is the execution frequency of basic block bb_j and δ_j is the increase in the critical path of basic block bb_j.

Ultimately we aim to synthesize a CDFG with smaller area via sharing. However, we need to insert steering logic into basic block datapaths to achieve this. Each multiplexer component mux_i will introduce an additional delay of d_i into the datapath. This delay can cause the critical path of a basic block to increase and hence the expected latency of the CDFG.

4.1 On the Complexity of Global Resource Sharing

It can be shown that the GRS problem is NP-Hard by transformation of an arbitrary Knapsack Problem instance to a GRS instance in polynomial time. We omit the details of this proof due to space considerations.

5. GIBBS: GLOBAL INTER BASIC BLOCK RESOURCE SHARING

In this section we present five heuristics to solve the GRS problem. All heuristics require an initial datapath for each basic block. In addition, they use a library of modules annotated with delay and area estimates. Note that the modules for a basic block might already be shared among different operations of that individual block. In addition, all heuristics are iterative algorithms. In each iteration only two resources from different basic blocks are merged together.

Let us first define the *criticality* of a resource. The idea behind criticality is to establish a relationship between the time spent to complete an operation and its effect on the overall execution time of the application. Specifically, we measure the criticality as

$$cr = \frac{basic\_block\_frequency}{slack\_of\_operation}$$

$basic\_block\_frequency$ is the execution frequency of the basic block containing the operation and $slack\_of\_resource$ is the slack of the operation in the initial schedule. Using the slack of the operations, GIBBS calculates the criticality of a resource. If multiple operations are assigned to a resource, the criticality of the resource equals to the maximum of the criticalities of the operations. Criticality is used by all heuristics to estimate the negative effects of resource sharing.

The first heuristic (**Heu-I**) tries to minimize the number of connections between the modules. This is achieved by examining the input and output connections of each module. During this examination, Heu-I determines a pair of resources with the most number of common inputs and outputs. To break ties, Heu-I considers the criticality of the resources and selects the least critical pair of resources. The least critical pair is the pair with the smallest *cr* value for the resource having the larger *cr* within the pair, i.e., minimum of maximums.

The second heuristic (**Heu-A**) pursues area minimization aggressively. Heu-A considers the estimated area gain for each sharing and selects the pair with the highest expected area reduction. Similar to Heu-I, ties are broken by the criticality rule.

The third heuristic (**Heu-P**) approaches the interconnect optimization by trying to capture common chains of resources with direct data communication. Consider an application with a frequent case of addition operation followed by a multiplication operation in several basic blocks. Heu-P checks the entire application and captures such common source-destination pairs. Once such combinations are found, they are prioritized to be shared as a chain of modules with other symmetric chains in different basic blocks.

The fourth heuristic (**H-S**) is based on criticality of resources defined earlier as an auxiliary measure for the previous heuristics. The mobility of operations assigned on a resource is an indication on how much extra delay can be tolerated along the path through that resource. This in turn, signifies opportunities to perform resource sharing such that the resulting increase in the path delay due to multiplexers can be absorbed within the operation mobilities without worsening the overall performance.

The last heuristic (**Heu-C**) combines Heu-I, Heu-A, Heu-P, and Heu-S. When comparing pairs of modules, Heu-C calculates the gain function for all three schemes. Then, it combines them in a weighted sum and merges the resource pair with the highest combined gain. For example, the gain for Heu-A is the expected fraction of area reduced by performing the sharing. Similarly, the gain of Heu-I is the fraction of interconnects avoided with resource sharing. The gain function for Heu-P is the avoidance of extra delay due to multiplexer insertion between consecutive modules in a chain. Specifically, the gain is the delay of the multiplexer that with other techniques would have been used, divided

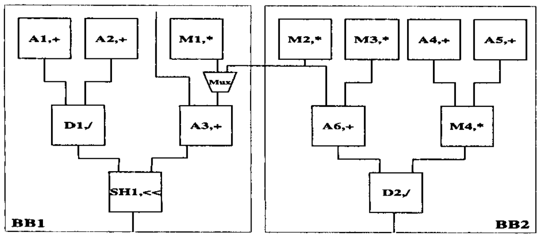


Figure 2: Initial datapath synthesized for a sample CDFG consisting of two basic blocks.

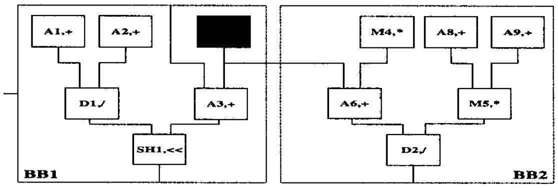


Figure 3: Datapath after applying one iteration of Interconnect-driven Heuristic Heu-I. Modules M1 and M2 are merged.

by the length of the critical path of the corresponding basic blocks.

The completion criteria for all heuristics are determined with a user-defined constant limit on the estimated extra delay. When each heuristic selects resources to be shared, it calculates the estimated increase in execution delay (using the multiplexer delays and the execution frequencies of the basic blocks). If the increase is above the threshold (Δ in Section 4), then the resource sharing halts without merging the last candidate pair. The iteratively transformed (partially merged) datapaths of the basic blocks are combined in a single CDFG datapath for generation of the final RTL VHDL by the high-level synthesis tool.

Let us demonstrate how our heuristics that deal with the input circuit topology, i.e., either target interconnect and/or area, perform resource sharing with an illustrative example. Consider the datapaths of two basic blocks (BB1 and BB2) depicted in Figure 2. They represent a portion of a larger design, and inputs/outputs at their interfaces are omitted for simplicity. When we apply Heu-I on these datapaths, resource pairs (M1,M2) and (A3, A6) are recognized to have the most number of common connections. M1, and M2 have the same destination, whereas A3, and A6 have one common source, which means they both share a common neighboring resource. Note that, there is a direct connection between M2 and A6. There is another connection between M2 and A3 going through the multiplexer module. Although there is a multiplexer inbetween, we treat this connection as a direct connection between M2 and A3, from the perspective of data communication. Sharing, i.e., merging A3 and A6 will result in a single connection between M2 and the merged module going through the multiplexer. The direct connection between M2 and A6 will be eliminated. At this point to break the tie between the pairs (M1, M2) and (A3, A6), Heu-I compares max{cr_M1, cr_M2} and max{cr_A3, cr_A6}. Assume, max{cr_M1, cr_M2} > max{cr_A3, cr_A6}. Then, Heu-I decides to share M1 and M2. The resulting datapath is shown in Figure 3. Heu-A, on the other hand, checks the area of the modules. The divider modules will clearly result in the best area reduction. Hence, Heu-A selects to merge D1 and D2 modules. The datapath after Heu-A is depicted in Figure 4. Note that, this decision forces two more multiplexers to be included on the critical paths of both basic blocks. Heu-P, on the other hand, counts the number of common pairs of connected resources. In Figure 2, there are four multiply-add, three add-divide, and two add-multiply chains. Hence, Heu-P will select mul-add pairs to be merged together if possible. M1 and M2 as well as A3 and A6 will be merged in a single iteration. The datapath after this sharing is shown in Figure 5. In our simulations, we put equal weight to all three heuristics within Heu-C. In this case, since the merging of dividers will result in significantly more area reduction than the other merging strategies, Heu-C will perform sharing of the dividers first. Figure 5 depicts the overall summary of the steps in the GIBBS technique.

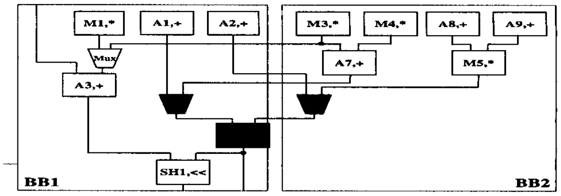


Figure 4: Datapath after applying one iteration of Area-driven Heuristic Heu-A. Modules D1 and D2 are merged.

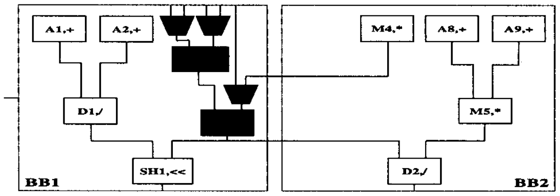


Figure 5: Datapath after applying one iteration of Pattern-based Heuristic Heu-P. Modules M1 and M2, and A3 and A6 are merged respectively.

6. RESULTS

To evaluate the effects of resource sharing, we have created an automated path from programs written in C to RTL

```
GIBBS(initial datapath for CDFG, module library, block_execution_frequencies, $\Delta$,
        heuristic-SEL ={Heu-I, Heu-A, Heu-P, Heu-S, Heu-C})

if (heuristic-SEL == Heu-A)
    place module types used in the CDFG datapath in a sorted list according to their area
    while (exec_delay_increase $\leq \Delta$) do
        if (list not empty)
            module type $T$ = head of sorted list
            module-pair($m_i, m_j$) = execute Heu-A(module library, $T$, CDFG datapath, block_execution_frequencies)
            if (module-pair($m_i, m_j$) == NULL)
                remove head of sorted list of types
           else
                merge ($m_i, m_j$) by combining the operations assigned to either resources on a single module
                insert multiplexer modules at the input pins of the merged modules if necessary
                update datapaths for affected basic blocks
        else
           return
        exec_delay_increase += Expected_Delay_Increase(module library, basic block pair ($b_i, b_j$), $f_i, f_j$)
    end while
    undo last resource sharing move
else
while (exec_delay_increase $\leq \Delta$) do
    module-pair($m_i, m_j$) = execute heuristic-SEL
    merge ($m_i, m_j$) by combining the operations assigned to either resources on a single module
    insert multiplexer modules at the input pins of the merged modules if necessary
    update datapaths for affected basic blocks
    return
exec_delay_increase += Expected_Delay_Increase(module library, basic block pair ($b_i, b_j$), $f_i, f_j$)
end while
undo last resource sharing move
```

Figure 6: Overall execution of the GIBBS technique.

VHDL. We use 9 MediaBench [1] applications for our experiments. The SUIF compiler infrastructure is used to perform the compiler optimizations and generate the IR, which is then transformed into a CDFG. We annotate the CDFG with basic block execution frequencies obtained through profiling. We have implemented our own high-level synthesis tool to perform the initial scheduling and binding for each basic block, which assumes that each instruction type in the CDFG can be mapped to a distinct module type. The initial scheduling and binding minimizes the number of modules required for each basic block. However, it does not perform any resource sharing among basic blocks. Then, separately we apply our resource sharing algorithms. Generation (and removal) of necessary modules while resource sharing is performed within our tool chain. After the resource sharing is completed, the tool generates an RTL VHDL description of the datapath for the complete CDFG including the insertion of necessary register and multiplexers. This VHDL code is then synthesized using Synplify Pro 7.0 from Synplicity and placed and routed by Xilinx Design Manager. We obtain area, wirelength and delay information after physical synthesis.

We have performed two sets of experiments. In the first set of experiments, we measure the effects of resource sharing on the area and execution delay of the design. The results for area are summarized in Figure 7. Overall, we see that the Heu-S and Heu-C perform the best, reducing the area by as much as 58% (42% on average) and 59% (44% on average), respectively. For most benchmarks, Heu-P did not bring significant improvement. However, for the gsm benchmark, it reduces the area by 24%. For the gsm application, we have combined the basic blocks from the gsm_decoder and gsm_encoder programs in one CDFG. These programs exhibit similar structures, hence Heu-P is able to capture large amounts of symmetry.

The effect on the execution time for all the heuristics is presented in Figure 8. We see that, for most benchmarks, resource sharing increases the delay. However, in some benchmarks we observe the benefit of having reduced the overall area, hence the wirelenght. We see that the Heu-S algorithm causes the largest increase in the delay. The reason lies in the fact that, Heu-S is the heuristic that performs the most resource sharing without considering the effects on the interconnect. In other words, Heu-S replaces large blocks with several smaller, but highly connected modules. Therefore, in most benchmarks it has a negative overall impact.

We were also interested in the relationship between the aggressiveness of resource sharing and delay penalty. To explore this, we have performed several experiments with the convolve benchmark varying the Δ value. Figure 9 summarizes the results for reduction in area as we change Δ value between 1% and 100% of the initial delay value. We see that, most heuristics increase their area efficiency as we allow larger delay penalties. The Heu-S, on the other hand, is not effected by the Δ value. The reason lies in the composition of the initial design. All heuristics estimate the area improvement for a sharing decision. Since extra multiplexers should be added to perform sharing, in practice only larger blocks (e.g., multipliers, dividers, adders) are considered for sharing. Therefore, for each benchmark, there is a hard limitation on the possible area reduction.

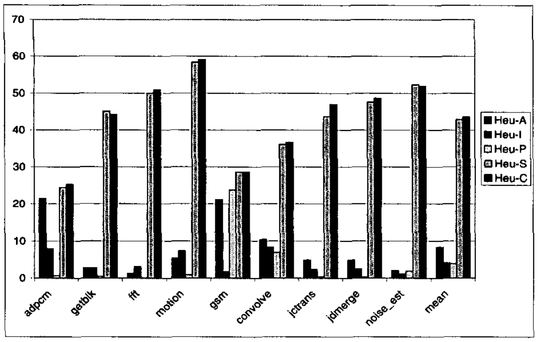


Figure 7: Reduction in area using resource sharing.

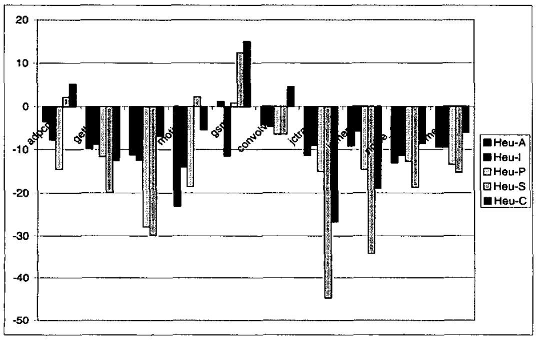


Figure 8: Increase in execution time due to resource sharing.

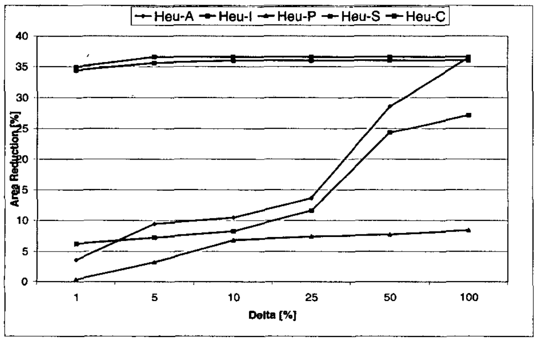


Figure 9: Effect of delta value on area improvement for convolve application.

7. CONCLUSIONS

In this paper, we presented a technique to perform global resource sharing for automatic synthesis of CDFGs. First, we proved that the global resource sharing problem is NP-Complete. Next, within the GIBBS framework we developed five heuristics. Each targeted a different aspect of resource sharing, e.g. number of connections, estimated area reduction, estimated increase in latency, etc. We synthesized nine benchmark applications from the MediaBench suite within our framework. Applying our global resource sharing strategies we were able to reduce the area of designs by 44% on average (using the criticality-driven heuristic) by allowing a 6% delay penalty.

8. REFERENCES

[1] W. H. Mangione-Smith C. Lee, M. Potkonjak. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems, November 1997. International Symposium on Microarchitecture.

[2] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler, April 2000. IEEE Computer.

[3] T. J. Callahan and J. Wawrzynek. Instruction level parallelism for reconfigurable computing, September 1998. Field-Programmable Logic and Applications, 8th International Workshop.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1999.

[5] M. B. Gokhale and J. M. Stone. Napa c: Compiling for a hybrid risc/fpga architecture, April 1998. IEEE Symposium on Field-Programmable Custom Computing Machines.

[6] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. A system for synthesizing optimized fpga hardware from matlab, November 2000. IEEE International Conference on Computer Aided Design.

[7] J. Hammes, R. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge. Cameron: High-level language compilation for reconfigurable systems, October 1999. Conference on Parallel Architectures and Compilation Techniques.

[8] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu. A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach. *TCAD*, 13(4):425–438, 1994.

[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock, 1992. International Symposium on Microarchitecture.

[10] P. Moisset, J. Park, and P. Diniz. Very high-level synthesis of control and datapath structure for reconfigurable logic devices, October 1999. Workshop on Compiler and Architecture Support for Embedded Systems.

[11] S. Raje and R. A. Bergamaschi. Generalized resource sharing, November 1997. International Conference on Computer-Aided Design.

[12] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast design space exploration in fpga-based systems, June 2002. ACM Conference on Programming Language Design and Implementation.

[13] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Gosh. Prism-ii compiler and architecture, 1993. IEEE Workshop on FPGAs for Custom Computing Machines.

[14] A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit, 2000. IEEE Symposium on Computer Architecture.