

Design Planning in Hardware Compilers

Seda Ogrenci Memik
Electrical and Computer Engineering
Northwestern University
seda@ece.northwestern.edu

ABSTRACT

Along with the fast evolution of programmable and reconfigurable systems, hardware compilers also advanced towards enhanced capabilities and optimization power. The responsibility of early hardware compilers was to map a subset of procedures or time critical computations onto hardware, where reconfigurable hardware was mostly utilized as a co-processing unit. A new generation of hardware compilers are needed to create frameworks for mapping significantly complex applications onto programmable devices. This can be achieved by empowering hardware compilers with new methodologies to increase their efficiency. Design Planning is one such concept. In this paper, we introduce techniques to enable a smooth, non-restrictive interaction between different compilation stages. These techniques are collectively referred to as Design Planning.

KEY WORDS

Reconfigurable Systems, Hardware Compiler, High-level Synthesis

1 Introduction

Today's high-end Field Programmable Gate Arrays (FPGAs) are large and sophisticated processing units capable of implementing System-on-a-Chip (SoC) designs. Such designs are called System-on-a-Re-programmable-Chip (SoRC). Designers are now able to use these devices to implement complex circuitry. Some of these FPGAs are equipped with embedded microprocessors and/or dedicated ASIC macros and IPs. Generating designs targeting such highly evolved devices presents new opportunities and challenges. Traditionally, designs were mapped onto FPGAs manually. However, with changes in the nature of designs and the increasing complexity of FPGA hardware, the manual process is becoming cumbersome. Consequently, starting from an elevated level of abstraction and automating the mapping process presents an attractive alternative. This process is also referred to as hardware compilation.

Two universal goals are pursued during the development of current hardware compilers for programmable systems. Incorporation of effective optimizations into the process and ability to generate feasible solutions with fewest number of iterations possible throughout the process. Optimizations can target a wide variety of goals, which can

be broadly considered as efforts to increase the quality of the final design in one or multiple aspects. Area requirement, execution time, power consumption and reliability can be named among others. The second goal, i.e., ability to quickly generate feasible designs and reach closure soon, particularly gains importance with the unique characteristics of the programmable systems. The most defining feature of programmable systems is the fact that they are pre-characterized and pre-fabricated. Their capacities in terms of logic resources are well-defined and their interconnect resources are also limited and discreet by nature. As a result, they are susceptible to designs with high logic and interconnect resource demand. As these devices are being used for implementing increasingly complex designs, the issue of feasibility is becoming more and more important.

Design Planning is our proposed approach towards creating more powerful compilation targeting programmable systems. The main philosophy behind Design Planning is early on recognition and management of relevant design metrics to create maximum freedom as compilation proceeds from one stage to another. Design Planning can be applied at various points throughout the synthesis flow at different boundaries separating different hardware synthesis tasks. As the synthesis flow for programmable systems is becoming more and more complex, the interaction between different subtasks becomes more crucial as well. Preventing a synthesis step from overconstraining the stages that follow is one important aspect of managing this interaction. Design Planning aims to leverage this by enabling the maximum freedom to be handed from one stage down to later stages.

The general concept of Design Planning can be materialized in different forms for various design metrics at different levels of the synthesis flow. In this paper, we will exemplify two applications of Design Planning onto hardware compilation for programmable systems. The first focuses on management of time slack within a schedule during RTL synthesis which in turn is exploited to create additional freedom for selection of resources and IPs during resource binding. The second application that we will discuss deals with management of resource sharing in order to pass designs with higher possibility of feasible placement and routing during the physical design stage. We will discuss techniques incorporated into RTL synthesis stage of hardware compilation. These techniques enable better coupling between different

sub-tasks in the flow.

The remainder of this paper is organized as follows. Section 2 gives an overview of existing approaches for hardware compilation targeting programmable systems. Section 3 presents the concept of Design Planning and discusses its applications to the RTL synthesis stage of hardware compilation. Section 4 concludes the paper with a summary and conclusions.

2 Overview of Existing Flows

There are several proposed methods and tools to create an automatic path from high-level design descriptions to programmable hardware. Early efforts to this end dealt with extracting computationally intensive kernels as 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. Compilers accompanied by their respective novel processor architectures like PRISM [1], Garp [2], NAPA C compiler [3], and Chimaera [4] were proposed.

Other projects proposed automatic datapath generation from algorithmic level descriptions. Such flows can be broadly categorized into two. Those that employ an internal RTL synthesis step and others that first transform application code into a behavioral hardware description and then utilize an external behavioral synthesis tool. Alternative approaches to hardware compilation are summarized in Figure 1.

Common to all flows is the initial compilation stage. In this stage the input program is transformed into an internal representation and optimizations are performed to leverage the transition from the algorithmic description to hardware. Many of those transformations and optimizations are inspired from their counterparts in software compilation. We can name constant propagation, redundancy elimination, function in-lining, and loop unrolling among others. Hardware-driven optimizations specifically targeting programmable systems have also been recently incorporated into such compilation steps. Kastner et al. proposed techniques for simultaneous template generation and extraction targeting hybrid programmable systems [5]. Templates extracted by their method are used to determine a set of hard macros or IPs that should be included within a hybrid programmable system that is customized for a specific platform. Kaplan et al. also proposed interconnect optimizations at the compiler level using a modified algorithm for Single Static Assignment (SSA) [6].

DEFACTO [7] is a compiler directed design environment for FPGAs to perform automatic hardware synthesis for loops. An external industrial high-level synthesis tool is used for the actual generation of datapath and control in this environment. Another hardware compilation framework that is coupled with an external behavioral synthesis tool is developed by Kastner [8].

The third group of hardware compilers perform an internal hardware synthesis although they may not necessarily make use of all high-level synthesis steps. The SA-

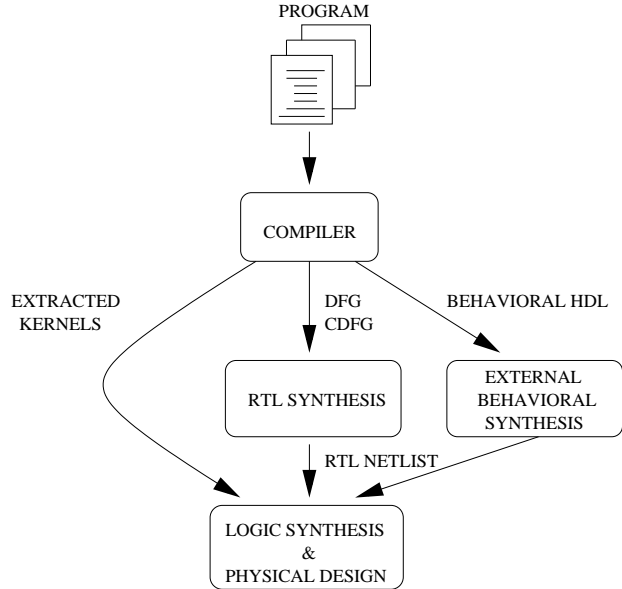


Figure 1. Approaches to hardware compilation targeting programmable systems.

C compiler [9] transforms programs written in a single-assignment subset of C into VHDL descriptions. The PipeRench compiler generates pipelined designs targeting the special PipeRench architecture [10]. HCircal [11] maps applications described in the high-level language Circal to reconfigurable hardware. The Circal language is designed to describe applications in sequences of processes. The hardware compiler maps each process to dedicated logic blocks. The NENYA hardware compiler [12] maps DFGs extracted from a Java program onto FPGAs. Cardoso [13] proposed a method for temporal partitioning and simultaneously allowing resource sharing within each temporal partition. Match [14] is a compilation environment to synthesize efficient hardware from applications written in MATLAB targeting reconfigurable systems. Celoxica offers the DK Design Suite for direct implementation of C-based algorithms into FPGA devices [15].

3 Design Planning during RTL Synthesis

As hardware compilers are becoming increasingly complex containing tasks from different domains such as those adopted from traditional software compilation domain as well as high-level synthesis tasks, achieving an efficient interaction between these steps becomes challenging. Design Planning is a methodology that can be incorporated into hardware compilation to address this issue. Design Planning collectively refers to techniques that enable early management and planning for design metrics which will allow for different compilation steps to interact in better coupling without overconstraining each other. We will demonstrate realization of Design Planning using a framework consist-

ing of an initial compilation stage which generates a CDFG representation of an application followed by an RTL synthesis stage, where scheduling of operations, resource selection and binding is performed. Specifically, we will focus on two Design Planning techniques. The first is incorporated into the scheduling stage within RTL synthesis managing time slack within a schedule in order to leave enhanced freedom for the later resource selection and resource binding steps. The second technique aims to provide a smoother coupling between RTL synthesis and physical design. In the following sections we will provide a detailed discussion of each of these techniques.

3.1 Planning for Time Slack during Scheduling

Time slack within a schedule is the amount of extra delay an operation can tolerate without violating any dependency constraints. Total available time slack within a schedule is a function of the given timing constraint, resource availability, and distribution and topology of dependencies among the operations that are scheduled. Although the total amount of time slack available within a schedule is a pre-determined value based on the aforementioned factors, the distribution of this total available slack among individual operations is determined by the scheduling decisions. By incorporating the notion of slack management within a scheduling algorithm a planned distribution of time slack to operations can be achieved. In turn, early planning for time slack can have an impact on the later synthesis stages.

The concept of slack is known to existing scheduling methods. Variations to Force-Directed Scheduling algorithms utilize the difference between ALAP and ASAP schedules of operations as a measure of mobility and then use this as means to prioritize operations during scheduling. However, those algorithms do not specifically try to reach a desired distribution of time slack to operations in the final schedule. We extended a scheduling algorithm proposed by Memik et al. [16] in order to incorporate planning for slack into the scheduling decision. This algorithm basically schedules DFGs in an iterative fashion, one path spanning the DFG at a time. Paths covering the input DFG are ordered according to their lengths and the algorithm starts with scheduling the longest path. Scheduling a single path is done by performing a non-crossing bipartite matching between operations along that path and feasible control steps to which operations can be assigned to. This is a weighted matching where the weight of each possible matching reflects the quality of the local scheduling decision. Optimization objectives such as latency and resource usage have been incorporated into the weight of each matching such that a matching maximizing the weight sum is the most desirable.

We can incorporate the slack distribution objective into the existing weight function of this scheduler in the following manner. Given a latency constraint λ , we evaluate the weight of a feasible matching for each operation considering the amount of slack that the operation can attain after this matching. Specifically, we aimed to distribute slack along paths homogeneously at each step. For this purpose, we first determine the average slack each operation can have

along a path. This is found by dividing the total slack along a path to the number of operation on the path. Then, while computing the weight for a certain assignment of an operation to a control step, we determine the difference between the average slack the operation could attain and the amount it can attain if the current assignment were to take place. The latency constraint is necessary in this case, since the flexibility in finish times of operations needs to be bounded.

The distribution of time slack among operations can be done in different ways depending on the ultimate goal. One possibility is described above, which is distribution of time slack among as many operations as possible in a homogeneous fashion. Alternatively, a certain type of operations can be selected and available slack can be distributed to those as much as possible. From the perspective of Design Planning our goal is to create a scheduling solution which will leave the highest flexibility for the following resource selection and resource binding stages. Many programmable systems allow use of IP libraries, and customized macros. Such elements are utilized towards various design objectives, such as increasing performance, reducing power consumption, reducing reconfiguration overhead, etc. Allowing extra delay on a selected set of operations during scheduling, alternative resources with varying delay characteristics can be selectively instantiated for a single operation. The current design objective might require replacement of a module with a slower instant. In that case, having planned for time slack on design objective-critical operations will allow the necessary freedom for the resource selection stage.

3.1.1 Results on Planning for Slack

In this section we present some experimental results with the modified scheduler in order to demonstrate how time slack can be manipulated in different ways with planning. First, we have used a weight function within the scheduler that only considers to minimize latency. This corresponds to the W/O Slack Objective case. Alternatively, we have added the slack component to the weight function and used the latencies obtained in the first case as our λ . By doing this, we are able to compare two schedules fairly. Table 1 presents the DFGs, the optimal latencies, the latencies obtained with the algorithm and the run times of the algorithm. Table 2 shows our results. The particular DFGs used for these experiments were chosen among popular DSP functions with suitable topologies to reflect improvement in slack. Availability of slack in a schedule depends on the topology of the input DFG as much as on the scheduling method. Therefore, in some DFGs it is not possible to see any effect of slack planning due to their structure. For the selected DFGs we have used two ALU resources and two multipliers. Each ALU has a delay of one clock cycle, and each multiplier has a delay of two clock cycles. The slack of operations is calculated as the difference between the control step when the result of an operation is ready and the control step at which the earliest scheduled successor of this operation demands the result. We report the number of operations that have non-zero slack values, i.e., the rest of the operations in the scheduled DFG had slack values equal to 0 or they were

I/O operations for which we do not report slack. We only report the slack values of arithmetic operations (MUL and ADD). We also give the total sum of the slack values on each operation type. We present a breakdown of these two measurements for the two types of arithmetic operations in the DFG, i.e. the ALU operations and multiplications (abbreviated as MUL in Table 2).

DFG	Optimal Latency	Our Latency	Runtime
ewf	28	28	10 msec
arf	18	20	7 msec
fir	16	16	6 msec

Table 1. Summary of DFG properties, latency and runtime results.

The results in Table 2 show that with proper planning to distribute slack on arithmetic operations our algorithm could indeed transform available flexibility in a schedule into additional slack for a specifically targeted set of operations; arithmetic operations in this case. For the arf benchmark we observe a degradation in slack value for ALU operations. The reason is due to the assignment of priorities for the two operation types ALU and MUL. The sensitivity towards increase of slack for MUL operations was set higher for these experiments. Therefore, the gain of increasing slack for MUL operations is evaluated to be larger than for ALU operations leading to greater tendency to allocate slack for MUL operations than for ALU operations. In the case of ewf due to the DFG topology and the order in which operations were scheduled MUL operations could not attain any slack whereas ALU operations could gain larger slack using the slack objective. For the fir benchmark slack objective affected the slack distribution for both operation types positively.

3.2 Global Resource Sharing

Many existing hardware compilers generate datapaths by creating dedicated components for each individual operation. This eliminates the need for sophisticated scheduling and binding stages. Assuming infinite resources an ASAP schedule can be simply generated for the operations in the datapath. However, when mapping large and complex applications onto programmable systems we cannot rely on the assumption of infinite resources anymore. Such an assumption is likely to yield infeasible designs for target devices with a fixed amount of logic resources and similarly limited routing infrastructure. Therefore, a better planning on resource distribution needs to be performed before the synthesized datapath is passed onto the physical design stages. This can be achieved through resource sharing. 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 as well as create highly congested nodes in the routing structure. Programmable systems having limited area and discreet routing architectures, are susceptible to designs with

high interconnect resource demand. Therefore a systematic approach to resource sharing is required. In the following, we will describe a technique for resource sharing called Global Inter Basic Block Resource Sharing (GIBBS).

We assume a synthesis flow within the hardware compiler that generates datapaths for individual basic blocks in the application program and combines them with a control flow. When mapping applications with large numbers of basic blocks, the resource demand will be very high as well. Therefore, 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 interconnect 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.

We formulate the problem of Global Resource Sharing (GRS) to efficiently plan for resource re-use. GRS refers to reusing a functional unit within datapaths of multiple basic blocks. We define 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 f_i 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 .

Δ , the increase in the expected latency of a CDFG is equal to $\Delta = \sum_j f_j \times \delta_j$, 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 total delay on the critical path of a basic block to increase.

We developed a heuristic algorithm solve the problem of Global Resource Sharing. We call this algorithm GIBBS. The main idea behind GIBBS is to identify pairs of resources within a given set of datapaths, such that those two resources will be merged and the operations assigned onto both resources will be re-assigned onto the single merged resource. The initial set of datapaths corresponds to a design that is synthesized with no resource sharing. Progressively, we eliminate resources out of the initial set of datapaths. The GIBBS algorithm seeks guidance of several design metrics which collectively indicate the impact of a possible resource sharing on the overall design. The sharing move, which indicates the highest gain according to those metrics is taken at each step.

The three GIBBS metrics used correlate with possible reduction in logic area requirement, number of module to module interconnects, and criticality of a resource. 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 exe-

	ewf		arf		fir	
	W/O Slack Objective	With Slack Objective	W/O Slack Objective	With Slack Objective	W/O Slack Objective	With Slack Objective
Num. of ALU operations with non-zero slack	4	5	4	2	3	4
total slack on ALU operations	14	16	9	8	7	9
Num. of MUL operations with non-zero slack	2	2	7	9	4	5
total slack on MUL operations	3	3	23	26	6	8

Table 2. Incorporating operation slack into scheduling objective.

cution time of the application. Specifically, we measure the criticality as $cr = \frac{basic_block_frequency}{slack_of_resource}$, where $basic_block_frequency$ is the execution frequency of the basic block containing the operation and $slack_of_resource$ is the slack of the operation assigned to the resource 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. In other words, the slack of a resource is determined by the minimum of slack values of all operations assigned to the resource. Criticality relates to the expected impact of increasing an operation’s execution time. This increase will be caused by the additional multiplexer. The area reduction metric simply considers the area of each module as it is pre-characterized within the module library and indicates a gain correlating linearly with the area of the module under consideration. The metric indicating interconnect requirement determines the pair of resources with the most number of common inputs and outputs.

By using the metrics described the GIBBS algorithm iteratively transforms (partially merged) datapaths of the basic blocks and the resulting datapaths are combined in a single CDFG datapath for generation of the final netlist by the RTL synthesis tool. The completion criteria is given by a user-defined constant limit on the estimated extra delay. After each selection of resources to be shared, the estimated increase in execution delay (using the multiplexer delays and the execution frequencies of the basic blocks) is calculated. If the increase is above the given threshold, then the resource sharing halts without merging the last candidate resource pair.

3.2.1 Results on Resource Sharing

To evaluate the effects of resource sharing, we have created an automated path from programs written in C to RTL VHDL. We use three representative MediaBench [17] applications for our experiments. They are selected based on the number of basic blocks they contained, which in turn is an indication to the resource requirements of these applications. A small, medium, and large application in that sense is selected out of the suite. The SUIF compiler [18] infrastructure is used to perform the compiler optimiza-

tions 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. Consequently, we apply our resource sharing algorithm. 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. Table 3 shows the are requirement with and without resource sharing targeting different Xilinx Virtex chips. Using FPGAs of various capacities, ranging from Virtex XCV150 to Virtex XCV800, we observe the trend for resource requirements. It can be seen that while it is possible to use a certain model with resource sharing it is always infeasible to use the same sized FPGA without sharing and we are forced to use a larger chip.

4 Conclusions

In this paper, we presented techniques that are incorporated into representative hardware compilers targeting programmable systems. Categorized under the general concept of Design Planning, these techniques essentially aim to create better coupling between different stages within the hardware compiler. They specifically address optimization and feasibility issues relevant to programmable systems. The ultimate goal is to create a compilation flow, where actions taken at one stage reach the local optimization objectives successfully without narrowing down the exploration space of the later stages for their respective objectives. To take it even a step further, our intent is to equip individual steps with enhancements allowing them to plan and manage design metrics which will help increase freedom to pursue alternative solutions or guarantees to reach feasibility in later stages. We demonstrated how time slack can be managed during scheduling to help the following resource selection

	XCV150	XCV200	XCV300	XCV400	XCV600	XCV800
adpcm						
w/o sharing	135 %	101 %	76 %	48 %	34 %	25 %
with sharing	102 %	75 %	57 %	37 %	25 %	19 %
getblk						
w/o sharing	247 %	181 %	139 %	89 %	62 %	45 %
with sharing	141 %	103 %	79 %	51 %	35 %	26 %
noise_est						
w/o sharing	641 %	471 %	360 %	230 %	160 %	118 %
with sharing	314 %	230 %	176 %	113 %	78 %	58 %

Table 3. Resource requirements (in terms of percentage of available CLBs) for different Virtex chips.

stage obtain more alternatives. We also presented a technique to reach a feasible solution sooner or with smaller cost (by being able to use a smaller FPGA chip in this particular example) through planning for resource sharing.

References

- [1] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Gosh, "PRISM-II compiler and architecture", IEEE Workshop on FPGAs for Custom Computing Machines, 1993.
- [2] T. J. Callahan, J. R. Hauser, J. Wawrzynek, "The Garp Architecture and Compiler", IEEE Computer, April 2000.
- [3] M. B. Gokhale, J. M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.
- [4] A. Ye, A. Moshovos, S. Hauck, P. Banerjee, "Chimaera: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit", IEEE Symposium on Computer Architecture, June 2000.
- [5] R. Kastner, S. Ogrenci Memik, E. Bozorgzadeh, M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", International Conference on Computer-Aided Design, November, 2001.
- [6] A. Kaplan, P. Brisk, R. Kastner, "Data Communication Estimation and Reduction for Reconfigurable Systems", Design Automation Conference, June 2003.
- [7] B. So, M. W. Hall, P. C. Diniz, "A Compiler Approach to Fast Design Space Exploration in FPGA-based Systems", ACM Conference on Programming Language Design and Implementation, June 2002.
- [8] R. Kastner, "Synthesis Techniques and Optimizations for Reconfigurable Systems", PhD Thesis, Computer Science Department, University of California, Los Angeles, September 2002.
- [9] J. Hammes, R. Rinker, W. Bohm, W. Najjar, B. Draper, R. Beveridge, "Cameron: High-level Language Compilation for Reconfigurable Systems", Conference on Parallel Architectures and Compilation Techniques, October 1999.
- [10] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Reed Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", IEEE Computer, Vol. 33(4), April 2000.
- [11] O. Diessel, G. Milne, "Compiling Process Algebraic Descriptions into Reconfigurable Logic", IPDPS Workshop, April 2000.
- [12] J. M. P. Cardoso, H. C. Neto, "Fast Hardware Compilation of Behaviors into an FPGA-based Dynamic Reconfigurable Computing System", Symposium on Integrated Circuits and Systems Design, September 1999.
- [13] J. M. P. Cardoso, "A Novel Algorithm Combining Temporal Partitioning and Sharing of Functional Units", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2001.
- [14] M. Haldar, A. Nayak, A. Choudhary, P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from MATLAB", IEEE International Conference on Computer Aided Design, November 2000.
- [15] I. Page, "Constructing hardware/software systems from a single description", Journal of VLSI Signal Processing, Vol. 12(1), 1996.
- [16] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner, M. Sarrafzadeh, "A Super-Scheduler for Embedded Reconfigurable Systems", IEEE International Conference on Computer-Aided Design, November 2001.
- [17] C. Lee, M. Potkonjak, W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", International Symposium on Microarchitecture, November 1997.
- [18] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, December 1996.