Classification of Compiler Optimizations for High Performance, Small Area and Low Power in FPGAs

Emre Özer, Andy Nisbet and David Gregg

Department of Computer Science
Trinity College, Dublin

Abstract

*We propose a classification of high and low-level compiler optimizations to reduce the clock period, power consumption and area requirements in **Field-programmable Gate Array (FPGA)** architectures. The potential of each optimization, its effect on clock period, power and area and machine dependency is explained in detail.*

1. INTRODUCTION

Embedded applications (*wireless networking/communications*, *digital signal processing (DSP)*, *video/voice/image processing, bioinformatics and mathematical/scientific*) are applications that can carry out specific functions such as speech encoding, edge detection or pattern searching. Embedded processors are processors that are designed to achieve higher performance and reduce power in embedded applications. They are used in mobile computers, cellular phones, PDAs and will be used to make any device or apparatus connected to a network in the near future such as a toaster, a paradigm called **pervasive computing**. Similar to embedded processors, FPGAs are reconfigurable devices that can be reprogrammed as needed. They consist of multiple programmable logic blocks with an interconnection network between each block. An application such as a speech encoder customized for an FPGA can run much faster than running on a general-purpose computer.

The main focus of this study is to design a competitive optimizing compiler for FPGAs by utilizing previous parallelizing and instruction-level parallelism (ILP) compiler experiences. Parallelizing compilers are developed to parallelize programs for several processors running simultaneously. They mainly focus on medium/coarse-level code parallelism. Then, instruction-level parallelism or ILP compilers emerge to increase instruction-level or fine-grain parallelism for superscalar and *Very Long Instruction Word (VLIW)* processors. ILP compilers utilize the techniques developed for the parallelizing compilers because multiple functional units in VLIW processors can execute instructions simultaneously. Our compiler cannot only take advantage of traditional parallelizing and ILP compiler techniques but also can introduce novel FPGA-specific optimizations.

2. RELATED WORK

PRISM-II compiler from *Brown University* [4] takes a C application and converts into VHDL code for FPGAs with standard frontend optimizations. They do not use backend compiler optimization techniques. RaPiD-C from *University of Washington* [5] is a C-like parallel programming language developed for reconfigurable architectures. It requires the programmer to specify parallelism and

partitioning into hardware resources. DIL compiler from *Carnegie Mellon University* [6] uses single assignment language semantics to compile for reconfigurable architectures. It uses a high-level language called DIL, produces dataflow graph and performs traditional graph, logic optimizations and finally place and route the produced netlist. Our compiler focuses on aggressive optimizations to produce efficient portable code, instead of dedicating its horsepower to placement and route and logic optimizations. Raw from *MIT* [7] presents a framework that compiles C or FORTRAN code into Verilog targeting custom hardware. The Raw compiler optimizations focus on more partitioning, placement and route of memories and computation blocks. The BRASS Group from *Berkeley* [8] uses a single-issue MIPS general-purpose processor with a slave reconfigurable hardware. Only certain loops or procedures are executed in the reconfigurable hardware. Their compiler reads programs written in ANSI-C, does loop optimizations and produce a dataflow graph to be synthesized. Our compilation framework has a wider scope with instruction and data parallelism since the whole program will be translated into portable HDLs.

Table 1 *Previous compiler frameworks*

Institution	**Name**	**Input**	**Output**	**Reliance**	**Optimizations**
Brown U.	PRISM-II	C	VHDL	Compiler	Frontend
U. Washington	RaPiD-C	C-variant	Verilog	Programmer	Frontend
Carnegie Mellon	PipeRench	C,Java	Dataflow	Compiler	Frontend
MIT	Raw	C	Verilog	Compiler	Frontend,
Berkeley	BRASS	C	Dataflow	Compiler	Frontend&Backend
Los Alamos Lab.	Stream-C	C-variant	VHDL	Programmer	Frontend
UC, Riverside	SA-C	C-variant	VHDL	Programmer& Compiler	Frontend&Backend
UC, Irvine	SPARK	C	VHDL	Compiler	Frontend&Backend
INESC	Galadriel	Java	VHDL	Compiler	Frontend&Backend

Streams-C compiler developed by *Los Alamos National Laboratory* [9] produces VHDL code from C applications. It consists of annotations and library functions to the C language and the programmer is responsible for explicit parallelism, storage allocation and definition of bit-widths of variables. Hence, it does not do aggressive backend compiler optimizations when compared to out compilation framework. SA-C from *University of California* at *Riverside* [10] is a single assignment C-variant high-level language designed to exploit fine and coarse-grained parallelism. It takes an application written in the extended C language and produces VHDL code after performing backend optimizations and parallelizations. Although it does aggressive optimizations, the programmer needs to write the application in this C-variant language. Also, their compiler does not perform feedback and profile-directed optimizations. SPARK for *University of California* at *Irvine* [11] is a high-level synthesis framework that takes ANSI-C code, performs frontend and backend optimizations and produces VHDL code. Although aggressive optimizations are applied, SPARK does not have feedback and profile-directed optimizations. Galadriel [12] compiles from Java bytecodes to VHDL by applying traditional loop optimizations and dag scheduling techniques. Table 1 shows the main characteristics of these previous similar works.

3. COMPILER OPTIMIZATIONS

All optimizations and transformations can be grouped into three functional categories according to their intended goals: *Performance* (*clock-period and clock cycles)*, *power* and area as shown in Table 2. Performance is measured with two different metrics: the clock period and the clock cycle time. The clock-period of the FPGA is determined by the slowest logic stage and the clock cycle time is computed as the total number of clock cycles required to execute the program in the FPGA. The optimizations/transformations can also be classified into two other subgroups as *machine and language dependence/independence*. Machine-dependent optimizations take advantage of specific hardware existing in the Xilinx Virtex-II Pro FPGA architecture.

Table 2 *Classification of Optimizations*

Optimization	Area	Power	Clock Cycles	Clock Period
Strength Reduction & Elimination	↓	↓		
Height Reduction			↓	
Instruction Scheduling			↓	
Variable Renaming			↓	
Instruction Combining			↓	
Expression Splitting				↓
Non-critical Path Simplification			↓	
Resource Sharing	↓	↓		↓
Bit-width Reduction	↓	↓		↓
Loop Unrolling	↑	↑	↓	
Loop Peeling			↓	
Loop Iteration Parallelism	↑	↑	↓	
Procedure Inlining	↑	↑	↓	
Procedure Splitting			↓	
Storage Reuse	↓	↓		
Software Pipelining			↓	

3.1 Strength Reduction and Elimination

Categories

a- Goal(s): ***Area*** and ***Power*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Multiplication and division algorithms in hardware are not power-efficient and take up a large of amount of chip area. Strength reduction replaces costly operations such as multiplication, division, modulo with less expensive operations or simplifies them by using algebraic properties. A complete elimination of such expensive operations is investigated by using profile-driven analysis. For instance, the numerator in a division may end up having zero during the execution of program or may be a multiple of the denominator. In the former case, the result of the division will be always zero and will be constant in the latter case.

The strength **reduction** and **elimination** of such expensive operations will result in **area reduction**, and therefore **reduction** in **power** consumption. **Clock cycle time** may **decrease** in case of **strength elimination** but may **increase** if only **strength reduction** can be applied.

3.2 Height reduction

Categories

a- Goal(s): ***Clock cycle time*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Height reduction techniques can reduce the data dependence chain in a loop where the operands are recurrences of previous iterations. It uses algebraic properties of expressions such as associativity, distributivity and commutativity. Consider the statements in a loop:

Instructions	Cycle
(1) sum = a + b	1
(2) sum = sum + c	2
(3) sum = sum + d	3

It contains a chain of data dependences (or *height*) of length 3, since the second and third statements take as input the value of sum from the previous statements. However, this code can be rewritten without changing the meaning to:

Instructions	Cycle
sum1 = a + b , sum2 = c + d	1
sum = sum1 + sum2	2

This allows the first two statements to execute in parallel, and reduces the height to length 2.

Cycle-time is **reduced** due to the decrease in the **data dependence height**. However, the overall **clock period** may **increase** due to **parallel** execution of instructions. **Area** may increase because of additional temporary variable created to parallelize the code.

3.3 Instruction scheduling

Categories

a- Goal(s): ***Clock cycle time*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Instruction scheduling techniques for VLIW architectures can also be applied to FPGAs. Basically, the VLIW instruction scheduler, whether it does local or global scheduling, looks for data independent instructions that can run simultaneously for each clock cycle. The number of instructions that can be scheduled in the same cycle depends on the number of functional units existing in the underlying VLIW processor. Practically, this number is not more than 8 functional units. On the other hand, the number of instructions that can be executed in the same cycle in FPGA depends on the placement and route of these instructions in the FPGA. Theoretically, all instructions in a program can run in a single cycle in an FPGA assuming that they are all data independent. Hence, scheduling simultaneous instructions in an FPGA is limited by its physical size, not by the number of fixed functional units as in VLIW processors. So, instruction scheduling can be considered to be machine independent in the context of FPGAs as opposed to VLIW processors.

Clock cycle time decreases as the **critical path** of the application is **reduced** due to grouping instructions that can run in a single cycle but similar to height reduction, the overall **clock period** may **increase** due to **parallel** execution of instructions.

3.4 Variable Renaming

Categories

a- Goal(s): ***Clock cycle time*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Variable renaming removes **false data** dependencies such as *anti* and *output* dependencies in the program. Anti and output dependencies put unnecessary constraints on instructions to be executed simultaneously in VLIW processors. In the context of FPGAs, they result in serialization in FPGAs. This is because it may not possible to reorder instructions in the presence of such false dependencies. For instance, take this example:

Instructions	Cycle
(1) a = b * c	1
(2) d = a + e	2
(3) a = b - f	3

Instructions 1 and 3 have output dependencies and 2 and 3 antidependencies. So, these three instructions take three cycles to execute. However, it would be possible to execute this code in two cycles if variable *a* in instruction 1 can be renamed to, say *tempa* as shown below:

Instructions	Cycle
(1) tempa = b * c	1
(2) d = tempa + e	2
(3) a = b – f	3

After renaming, instruction 1 and 3 can be scheduled in the same cycle and the total execution time is reduced by one cycle.

Instructions	Cycle
(1) tempa = b * c, (3) a = b – f	1
(2) d = tempa + e	2

Clock cycle time improves with elimination of false data dependencies because the schedule height can potentially be reduced. Similar to the instruction scheduling, the overall **clock period** may **increase** due to **parallel** execution of instructions. **Area** increases due to extra renaming registers.

3.5 Instruction combining

Categories

a- Goal(s): ***Clock cycle time*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Instruction combining combines data dependent instructions into a single instruction to reduce the critical path. However, it may increase the overall clock cycle time if a large number of relatively expensive instructions are combined. Thus, instruction combining should be performed for fast and inexpensive operators. For instance, the sequence a=b&c; a=a&d, would normally take two cycles, since there is a data dependence from the first statement to the second. However, these statements could be rewritten as a=b&c&d; which can execute in a single cycle.

Clock cycle time decreases due to **reduction** in the **critical path**. **Clock period** may **increase.**

3.6 Expression splitting

Categories

a- Goal(s): ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

This is exactly the opposite of instruction combining. An instruction may have a long and complicated expression structure. When configured on an FPGA, such a complicated instruction may reduce the clock cycle time. For example, an expression like *"x = a * b + c / d"* is executed in a single cycle on an FPGA. However, it may reduce the overall clock cycle time if this expression is the slowest expression in the program. In this case, it is always better to split the expression into two more subexpressions. For instance, *"x = a * b + c / d"* can be split into three subexpressions: *"temp1 = a * b"*, *"temp2 = c / d"* and *"x = temp1 + temp2"*. Then, they will be scheduled and executed sequentially in the hardware.

The motivation behind expression splitting may seem contradictory with instruction combining. A cost function analysis must be performed whether expression splitting or instruction combining is the optimal for the given constraints. *Profile* and *Performance/Power/Area feedback data* will be analyzed to estimate these constraints and the linear equations will be formulated. The linear equations will be solved using *linear programming* techniques to decide whether an expression should be split or whether a group of instructions should be combined.

The overall **clock period decreases** due to the splitting of complicated expressions with a **cost** of **increasing** the **clock cycle time**.

3.7 Non-critical Path Simplification

Categories

a- Goal(s): ***Clock period*** *Decrease* and ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

It is vital that instructions on the critical path are executed as quickly as possible. However, in a typical program many instructions are not on the critical path. These instructions can be made as sequential as possible and simplified using expression splitting, strength reduction and elimination techniques in order to allocate them simple hardware in the FPGA. The execution time on the non-critical paths may increase but it should not pose any problem since they are not executed frequently. These simplifications on the non-critical paths will not increase **the clock period** either.

The critical path can be determined by the profile and performance feedback data. Most of FPGA real estate will be dedicated to the instructions on the critical path in order to **reduce** the total **clock cycle time** while **maintaining lower clock period**. The total **area** may or may not increase depending on how aggressively instructions on the critical path are optimized.

3.8 Resource Sharing

Categories
a- Goal(s): ***Area*** and ***Power*** *Reduction*
b- Dependencies: *Machine* ***Independent***

A common optimization in hardware description languages (HDLs) is resource sharing in order to reduce area and power. Frequently used operators such as addition, subtraction, multiplication and division are shared by several expressions in the program instead of instantiating separate hardware for each use of the operator.

Resource sharing **does not comprise performance** if resources are shared by instructions that are **not scheduled in the same cycle**. Otherwise, resource sharing will prevent instructions accessing the same resource from being scheduled in the same cycle. If the objective is to maximize performance, the trivial way of finding optimal shared resources is to find the common set of operators used by the instructions that are not executed in parallel. On the other hand, finding the optimal set of shared resources is an optimization problem if the objective is to minimize the total area usage while keeping the performance as high as possible. This is again an optimization problem that can be solved by using genetic algorithms, linear programming or other exhaustive search techniques.

3.9 Bit-width Reduction

Categories
a- Goal(s): ***Area***, ***Power*** *Reduction* and ***Clock period*** *Decrease*
b- Dependencies: *Machine* ***independent***

It is possible to declare variables and arrays with exact bit-widths in HDLs unlike high-level programming languages like **C** or **Java**. In practice, most of the bits in registers and memory locations are **never** used during the course of the program execution. For instance, if a variable that is declared as a 32-bit integer is used as a flag of zero or one values, then the rest of the bits are wasted in hardware. This causes an excessive increase in area and therefore power consumption.

Bit-widths can be calculated at compile time by performing data-flow analysis techniques. On the other hand, they can also be estimated by using profiling techniques and estimations will be transferred to the backend compiler.

Bit-width reduction causes significant reductions in area and power consumption. Potentially, it may decrease the **clock period** by shortening the wire lengths and the sizes of circuits due to reduction in bits.

3.10 Loop Unrolling

Categories
a- Goal(s): ***Clock period*** *Decrease*
b- Dependencies: *Machine* ***Independent***

A fully unrolled loop is a straight-line code in which each iteration of the loop body is sequentially laid out. Full loop unrolling gives great opportunities for instruction scheduling to group data independent instructions in the same cycle.

Handel-C allows *for* loops to be fully unrolled by replacing them with *seq* keyword. On the other hand, Verilog has no such a keyword, so the loop body has to be replicated by compiler.

However, it expands the code size, and therefore **increases** the **demand** for more **area** and **power**. The **execution time** of the program **improves** with full loop unrolling followed by instruction scheduling.

3.11 Loop Peeling

Categories

a- Goal(s): ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

Instead of performing expensive loop unrolling, loop peeling strips off a compile-time constant number of iterations and leaves the rest of iterations in the loop. Again, partial loop unrolling has to be followed by instruction scheduling.

Even though the code size increase is not as big with full loop unrolling, it still **increases area** and **power**. Similarly, the **execution time** of the program only **improves** if loop peeling is followed by instruction scheduling.

3.12 Loop Iteration Parallelization

Categories

a- Goal(s): ***Clock cycle time*** *Reduction*

b- Dependencies: *Machine* ***Independent***

Loop iteration parallelization examines if all iterations can be executed simultaneously. It does this by checking loop-carried data dependencies between iterations and it can only be applied to the loops whose bounds are compile-time constants.

Handel-C supports a keyword called *par* that can replace *for* keyword and then each iteration executes at the same time. On the other hand, Verilog does not have an equivalent keyword but is still possible to put each iteration under separate *always* statements and make them fully parallel.

This technique significantly **reduces** the **execution time**. However, it may increase the **clock period** due to parallel executions of iterations, especially if the number of iterations is high. For similar reasons, it also **increases area** and **power**.

3.13 Procedure Inlining

Categories

a- Goal(s): ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

Procedure inlining offers opportunities for extracting more instruction-level parallelism in VLIW processors. A VLIW compiler basically integrates the procedure to be in-lined at its call site. However, this technique can be utilized for FPGAs in two different ways. The first way is to use it as done by VLIW compilers in order to increase the instruction window for the instruction scheduler. The second way is to

use translated language features. In Handel-C, it is possible to declare a function with the *inline* keyword. In this case, separate hardware is instantiated whenever the procedure is called. On the other hand, functions and tasks in Verilog are always instantiated as separate pieces of hardware.

Procedure inlining **reduces** the **execution time** by creating separate logic for each call but it **significantly increases power** and **area** requirements. Also, it may put a **tight pressure** on the **clock period** if, for instance, two calls are to be executed at the same cycle.

3.14 Procedure Splitting

Categories

a- Goal(s): ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

Procedure splitting splits a procedure into two components of most frequently executed and less frequently executed parts. Potentially, the most frequently parts and less frequently parts can be made separate procedures. The most frequently parts can be inlined and this will decrease the cycle time by eliminating unnecessary area increase by not inlining the less frequently parts of the procedure.

3.15 Storage Reuse

Categories

a- Goal(s): ***Area*** and ***Power*** *Reduction*

b- Dependencies: *Machine* and *Translated Language* ***independent***

Storage reuse is similar to register allocation in general-purpose processors except that there is no fixed number of registers in FPGAs, and the storages that are candidates for reuse depend on the structure of the program. In the context of FPGAs, storages are flip-flops that have been allocated for variables, pointers and arrays. Storage reuse allows storage locations that are dead in the program to be reused by other variables, array locations or pointers. It is a useful technique when FPGA area is of major concern for the given application. The storage location in an assignment or in a definition can be considered dead after its last use as an operand.

The problem of storage reuse can be stated as follows: there is a set of dead storage locations that can be candidates for reuse and there is another set of live storage locations that are candidates to be replaced by dead storage locations. These two sets are disjoint sets. As such, the problem can be turned into register allocation problem as the dead storage locations can be considered as available registers and the live storage locations are allocatable objects. Another dimension is added to the problem since the widths of the storage locations may not be the same. This situation may not be a constraint however. For instance, two live storage locations of each 4-bit long can be assigned to an 8-bit dead storage location.

The disadvantage of storage reuse is that it creates extra multiplexers for each use of a storage location. Excessive storage reuse may therefore have a negative effect on area and should be done carefully. Once again, it can be represented by an optimization problem: to find the maximum set of reusable live storage locations while minimizing the area overhead incurred by the multiplexers.

Storage reuse potentially **reduces area** requirements and may therefore **reduce power consumption**. However, it may also **increase** the **execution time** by introducing new data dependencies between instructions due to variable reuses.

3.16 Software pipelining

Categories

a- Goal(s): ***Clock period*** *Decrease*

b- Dependencies: *Machine* ***Independent***

Software pipelining is a technique used in VLIW compilers and essentially performs instruction scheduling for loops. In addition to acyclic instruction scheduling, instructions can also move from one iteration of a loop to another. This allows instructions from different iterations of the same loop be grouped together for parallel execution. The following shows an example of scheduling and pipelining a loop to sum the elements of an array.

```
do {                    do {                    x = a[i];          // prologue
    x = a[i];               x = a[i];           do {
    sum = sum + x;          sum = sum + x, i = i - 1;     sum = sum + x, i= i -1, x=a[i];
    i = i – 1;              } while ( i != 0 );           } while ( i != 1 );
} while ( i != 0 );                                       sum=sum+x;   // epilogue
```

(1) No scheduling (2) Loop body Scheduling (3) Software pipelining

In the above example, each iteration of unscheduled loop (1) will take three cycles. However the critical path of the loop body is only two cycles, since the decrement of *i* can be executed in parallel with either of the other statements. The longest cyclic dependence in the loop is of length one, so we can produce the software pipeline (3), where all operations in the loop execute in parallel, reducing the length of the loop to one cycle. Note that to maintain the code semantics, the compiler has to add extra copies of some of the statements in the loop for the first and last iterations.

These are known as the *prologue* and *epilogue* of the loop. The size of the loop prologue/epilogue can often be greater than the size of the loop body using existing software pipelining algorithms. One approach to reducing the prologue/epilogue size is based on the observation that the prologue exists only because some instructions in the loop depend on values from previous iterations already being available. The compiler can *if-predicate* these instructions to prevent their execution on the first iteration(s) of the loop. The advantage of this is that it can eliminate the loop prologue. The disadvantage is that it adds extra complexity to the body of the loop, which may affect the clock period. A similar approach can be used to eliminate the epilogue. Another alternative would be to allow these statements to execute, but ensure that the input data they read has no effect on the overall computation, and that any values they write to variables are never used.

In Handel-C, it would be possible to implement software pipelining of a loop. In a parallel loop with *par* keyword, a regular hardware pipeline structure can be instantiated and epilogue and prologue instructions can be selectively executed by using *ifselect* keyword. Basically, *ifselect* compiles instructions if a compile-time

constant expression is evaluated as true. Otherwise, the *else* block is compiler. It can be used to compile epilogue and prologue instructions for the first and last few iterations assuming that the number of the first and last few iterations is know at compile time.

Software pipelining certainly **reduces** the **execution time** and may **not** put much pressure on the **cycle period** if carefully implemented. Increase in **area** due to prologue and epilogue instructions will **not** be significant.

3.17 Utilization of On-chip Block RAM and ROM

Categories
a- Goal(s): ***Area*** *Reduction*
b- Dependencies: *Machine* ***Dependent***

Xilinx Virtex-II Pro FPGA architecture has on-chip block RAMs and ROMs. The backend compiler can assign some of the arrays to these available on-chip RAMs and ROMs so that a lot of logic on the FPGA can be allocated to other parts of the program.

The benefit from using on-chip block RAM/ROMs is to **save area** from configurable logic for arrays.

3.18 Utilization of On-chip Multipliers

Categories
a- Goal(s): ***Area*** and ***Clock period*** *Decrease*
b- Dependencies: *Machine* ***Dependent***

Xilinx Virtex-II Pro FPGA architecture has also on-chip 18-bit multipliers that the backend can take advantage of. They can also be tied together to make bigger multipliers. These can be assigned to some of multiplication operators used in the program, especially with the multiplication instructions that are not on the critical path.

Similar to using block RAM/ROMs, using on-chip multipliers **saves area**.

4. CONCLUSION

This study investigates classification and the potential benefits of frontend and backend compiler optimizations for applications targeting FPGAs. The classification is made whether the optimization reduces or increases one of the three metrics: clock period, power consumption and area in the FPGA.

REFERENCES

[1] R. P. Wilson, R. S. French, C. S. Wilson, S. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *Tech. Report*, Computer Systems Laboratory, Stanford University, CA, USA, 1994.
[2] Xilinx, http://www.xilinx.com

[3] Celoxica, *Handel-C Language Reference Manual*, Version 3.1, 2002.
[4] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. "PRISM-II Compiler and Architecture", *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, April 1993.
[5] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD", *Field-Programmable Custom Computing Machines*, 1998.
[6] M. Budiu, and S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics", *7th ACM International Symposium on Field -Programmable Gate Arrays*, 1999.
[7] Jonathan Babb, Martin Rinard, Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe, "Parallelizing Applications Into Silicon", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines '99* (FCCM '99), Napa Valley, CA, April 1999.
[8] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler", IEEE Computer, April 2000.
[9] J. Frigo, M. Gokhale, and D. Lavenier "Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective", *9th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February, 2001.
[10] B. A. Draper, A. P. W. Böhm, J. Hammes, W. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, J. Bins, "Compiling SA-C Programs to FPGAs: Performance Results", *International Conference on Vision Systems*, Vancouver, July, 2001.
[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations", *the 16th International Conference on VLSI Design*, New Delhi, India, Jan. 2003.
[12] J. M. P. Cardoso, and H. C. Neto, "Towards an Automatic Path from Java Bytecodes to Hardware Through High-Level Synthesis", *Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems*, Lisbon, Portugal, Sep. 1998.