On Estimations for Compiling Software to FPGA-based Systems*

João M. P. Cardoso[1,2]

[1] University of Algarve / Faculty of Sciences and Technology
Campus de Gambelas, 8000 – 117 Faro, Portugal
[2] INESC-ID, Lisbon, Portugal
jmpc@acm.org

Abstract

This paper presents recent advances in a compiler infrastructure to map algorithms described in a Java subset to FPGA-based platforms. We explain how delays and resources are estimated to guide the compiler through scheduling and temporal partitioning. The compiler supports complex analytical models to estimate resources and delays for each functional unit. The paper presents experimental results for a number of benchmarks. Those results also arrise a question when performing temporal partitioning: shall we try to group as many computational structures in the same configuration or shall we have several configurations?

1. Introduction

Reconfigurable computing has already confirmed a significant potential for accelerating certain general-purpose computing tasks. Since reconfigurable systems are capable to provide a mix of flexibility, high-performance, and energy savings [1] (when compared to microprocessors and DSPs) their use is growing. However, as reconfigurable computing is based on hardware foundations, it inherited its design difficulties. The approaches used for the realization of the most successful applications using reconfigurable systems required user's hardware expertise and the capability to master the low level architectural details of the target reconfigurable device. Today's reconfigurable system devices, with millions of logic gates, distributed memories and embedded multipliers, are able to accommodate complex algorithms. Thus, one of the most challenging issues is the methodology to efficiently and automatically map computations (described in software programming languages) to those systems [2][3]. Moreover, compilation time should not be significantly higher than compilation to a microprocessor in order to permit rapid development and evaluation of different high-level decisions (e.g., hardware/software partitioning), both important to deal with time-to-market pressures. Although several approaches have been studied, implemented, and experimentally evaluated, software compilation targeting reconfigurable computing systems is still deviated from the desired success. Is it on the same track that conducted to the little success of high-level synthesis tools?

FPGA (Field-Programmable Gate Array) architectures are not easing the compilation task. Lack of commercial architectures closing the gap and helping the compilation has been one of the obstacles to push-button approaches. However, being each time more advanced and powerful, they enable high creativity solutions and efficient support structures for typical computations. One example is the use of the existent registers and memory blocks to cache data [4]. Certainly, further advancements will permit more opportunities to achieve high-performance solutions.

Even with long reconfiguration times, authors have shown that it can be hidden and global speedups can be achieved [5]. Since the execution on reconfigurable computing platforms can be divided in sequential stages (e.g., fetch, configuring, and execution [6]) it is important to find, from the input algorithm, the set of configurations that both produces feasible implementations (considering the available number of resources) and minimizes the overall performance considering the overlapping of the stages related to different configurations.

Bearing in mind some of those issues we are working on a compiler targeting FPGA-based platforms. In this paper we present our compilation environment, explain how estimation and temporal partitioning is being considered, and discuss some of the relevant issues in order that compilation of software programming languages to FPGA-based platforms fulfills the promise: "to bring reconfigurable platforms to software programmers".

This paper is organized as follows. In section 2 we briefly present our compiler infrastructure. The scheme for estimating resources and delays is sketched in section 3. In section 4 we present experimental results. Section 5 gives a discussion on topics related to compilation. In section 6 we present the related work and conclude in section 7.

2. Compilation of Software to FPGAs

Due to the popularity of C, especially to program embedded systems, compilation of C programs has been fo-

* This work has been partially supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002).

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

cus of many academic and industrial efforts [7]. Bearing in mind the increasing popularity of other languages and representations, some authors focus on compilation of other software programming languages (e.g., Java [8]) to FPGA-based platforms.

Aiming to solve some of the problems to map software programming languages to reconfigurable architectures we have been working on compiler techniques for FPGA-based platforms [9][10]. Our choice has been Java and particularly the java bytecodes representation. The compilation flow, depicted in Figure 1, is mainly supported by the Galadriel and Nenya compilers [9]. The compilers generate a specific architecture to execute the computational structures present in the Java code of an application[1]. The architecture is represented by a data-path and a control unit specified as a finite state machine (FSM) described in behavioral style. Recently, Nenya has been modified to generate XML (eXtensible Markup Language) representations of the data-paths, control units and reconfiguration transition flows (represented by an RTG – reconfiguration transition graph) [11]. Using XSL (eXtensible Stylesheet Language), the representations can be transformed to VHDL and other suitable languages or formats. The use of XML representations enables users to define their own XSL translation rules to output representations using the chosen language (e.g., Verilog, VHDL, SystemC, etc.).

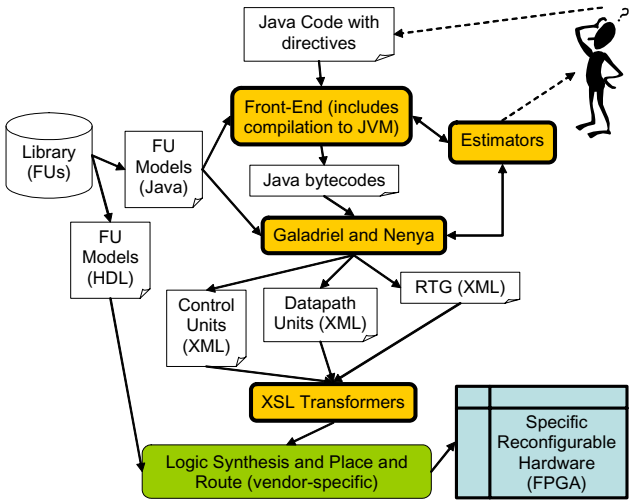


Figure 1. Compilation flow.

The front-end is currently a java to bytecodes compiler (e.g., javac from Sun Microsystems Inc.). We have plans to extend this with a front-end able to perform a number of optimizations, e.g. loop unrolling, at the source code level based on user directives. The estimators are at the moment able to calculate estimations of the number of resources required to implement the code being compiled to the target FPGA. Future plans include high-level estimations to acquire the impact of code transformations, e.g. to decide about loop unrolling [12].

The compiler generates a centralized control unit for each temporal partitioning. The control unit generation is achieved using a scheduling of operations, based on delay estimations of the operations presented in the representation graph. For temporal partitioning we also need efficient estimation of resources so that the temporal partitioning step is automatically performed without the need of several iterations through the overall compilation flow (including logic and physical synthesis). Next section explains how estimations are conducted by the compiler.

When mapping sequences of loops a question may occur: shall we map all the loops using a large design and a single configuration or shall we split the program and generate several and smaller configurations? This decision may have impact on the overall performance[2], especially if reconfiguration time improves in future FPGAs. There might be a number of reasons for mapping the computational structures in more than one configuration:

- With small designs, the P&R tools may figure-out better usage of resources and each design may be clocked at higher frequencies;
- Small designs may need shorter interconnections and so interconnection delays may be smaller;
- Splitting the input program may also split accesses to the same memory and hence each design (configuration) may have simpler memory interface units and fastest accesses;
- Some reconfiguration time can be amortized by fetching or configuring concurrently with the execution of a certain configuration.

However, there are also a number of disadvantages when splitting a program:

- Pipelining of loops in sequence is not possible when those loops are in different configurations;
- Data communication between configurations may be needed. This is not a problem when the data to be communicated is stored in memories with or without temporal partitions and those memories maintain contents between configurations. When data must be stored to be loaded by subsequent configurations this may impose a communication overhead that must be amortized by the execution time of the configuration in order to lead to efficient solutions.

We use a temporal partitioning scheme working at the top level of the HTG (Hierarchical Task Graph) that tries to take into account such issues. At the moment we use a simple heuristic to merge HTG top level nodes in the same configuration. The scheme starts with each HTG top level node in a different temporal partitioning and then tries to

[1] The compilers accept a Java subset which includes loops, arrays, primitive types such as int, short, byte, etc.

[2] Previous work has shown this in the context of a coarse-grained array architecture: the PACT XPP array [6].

merge nodes bearing in mind the configuration hiding that may be achieved when the execution of one configuration can be concurrently performed with the reconfiguration of a subsequent temporal partition. The approach is based on our previous work [6].

3. Delay and Resource Estimations

Behavioral synthesis uses a library of functional units (FUs) to implement the operations existent in the high-level language, such as arithmetic, logic, and relational operations. To make easier the scheduling problem, some behavioral synthesis tools do not support models to take into account for the vast bit-width properties of the FUs to implement the computational structures. They usually include costs of typical bit-widths such as 8, 16 and 32 bits. In other cases, tables indicating a discrete number of bit-widths and the delay and number of resources for each one can be specified. Based on those discrete values, non-linear and linear regression can be used to obtain the analytical models. The use of linear regression to model FUs has conducted to satisfactory results as has been reported in [13].

In [8] we started using a representation model to take into account discrete number of input bit-widths and the use of simple equations to model resources and delays. However, for new FPGA devices resource and delay models can be very complex. To achieving the required model complexity, we included in the compiler the support of the main features of a programming language in order to specify complex delay and resource models for each FU.

Each FU has resource and delay models, based on the bit-widths of each operand and each output, as represented by equations (1) and (2), respectively. In equation (1) "resources[]" is an array that represents various type of resources in the target FPGAs, such as the number of 4-LUTs, the number of embedded multipliers (MULT18x18), the number of flip-flops (FFs), etc. In equation (2) "delay[]" is an array that represents the delay of the FU and the number of pipeline stages.

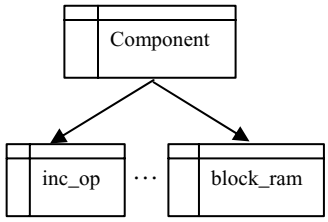
$$resources[\,] = f_{resources}(w_1, w_2, ..., w_N) \quad (1)$$

$$delay[\,] = f_{delay}(w_1, w_2, ..., w_N) \quad (2)$$

Besides the HDL description, there is a Java class for each FU in the library. An XML file is used to specify the name of the FUs and the correspondent operation (see Figure 2). Each class must extend a pre-defined abstract class (Component) and must implement two methods, *delay* and *resources*, which resemble the equations (1) and (2). The component class stores the bit-width for each input and each output of the FU after it is bound to a certain operation in the hardware graph (final graph that resembles each design and from it the final representation is directly obtained).

```
<LIBRARY>
<UNIT name="inc_op"
operation="iinc" />
<UNIT name="add_op"
operation="iadd" />
<UNIT name="mult_speed_op"
operation="imul" />
<UNIT name="mult_area_op"
operation="imul" />
…
<UNIT name="block_ram"
operation="memory" />
</LIBRARY>
```

(a)



(b)

Figure 2. Library of FUs: (a) library specification in XML; (b) each unit in the library needs a Java class with the same name and inheriting the abstract class Component;

Using the capabilities of the Java reflection API, a user can redefine the library models according to the target reconfigurable architecture and make them available for the compiler just by compiling the Java description of each FU. Complex models of resources and delays can be specified by using Java programming structures.

Consider the 3-D plot in Figure 3 as an example of a resources model for a multiplier FU (targeting a Xilinx Spartan-2 FPGA). One can see that the analytical models must have the capabilities to attain for the idiosyncrasies of each FU in the target FPGA. The problem is even more complex when targeting FPGAs with embedded multipliers. In some of the Xilinx FPGA families there are MULT18x18 distributed multipliers on the chip. This means that when the compiler needs a multiplier with less than 18 bits on each input operand one MULT18x18 is used. However, when input bit-widths are higher than 18 bits the multiplier is implemented with MULT18x18 blocks and additional 4-LUT resources. This example shows the need to support complex specifications for analytical models. In our case, since the models are specified in Java, we can use all the Java features such as if-then-else structures to model each FU in the library.

Although the process to generate the models for each FU uses Perl scripts that go through a pre-defined set of bit-widths, call the logic and physical synthesis tool, and output the discrete values for delays and resources, we bear in mind a fully automated process since each time we target a different FPGA family the process to acquire the models must be repeated accordingly. Based on the structure of each FU and on some characteristics of the target FPGA there should be feasible to estimate the number of resources and the delay of an FU without a "synthesis and measure" approach. Is it possible to calculate with enough accuracy the number of resources and the delay for each FU given the bit-widths, FPGA characteristics, and a high-level model of the FU structure? We plan to address this question in a near future.

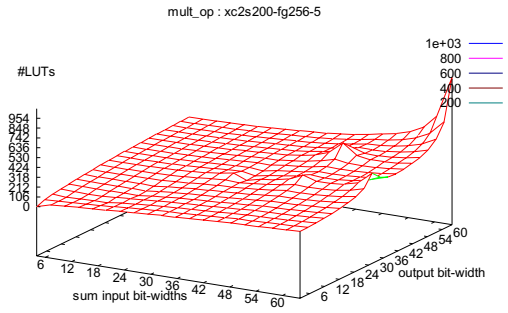


Figure 3. Number of 4-LUTs needed for mul_op FUs for different input/output bit-widths.

The estimation of the resources for each data-path is based on the simple addition of the resources for each FU in the hardware graph. The estimation of the resources for each FSM is based on the assumption that they are implemented using one-hot encoding. The number of flip-flops (FFs) is equal to the number of FSM states and the number of resources for the combinatorial logic of the FSM is estimated based on the states, input signals, output signals, and the state transition graph (STG) output by the compiler. For this we use an analytical model obtained after statistical analysis of the type of FSMs generated by the compiler for a set of examples.

4. Experimental Results

We have performed a number of experiments using our compiler. As benchmarks we use a Java version of the Fdct (C source available from Texas Instruments Inc.), a forward 2D wavelet based on the Haar algorithm (Haar), and two image processing kernels which resemble typical sequence of image processing tasks: a finite impulse response filter (FIR) and an edge detector (Sobel). Regarding compilation to FPGA-based platforms, these benchmarks can be considered of medium complexity. Table 1 shows the main characteristics of the examples. Number of loops, array variables, JVM instructions, basic blocks, lines of VHDL, and FUs used in the generated architecture are shown for each example. We also show characteristics of temporal partitions for each benchmark.

We use the Xilinx ISE 6.3i for the logic and physical synthesis of the designs output by the compiler and the Xilinx Virtex-2 (xc2v3000-bf957-5) as the target FPGA.

Figure 4 compares the number of resources between estimations performed at the compilation level over the resources output by the back-end tools, which include logic synthesis, mapping, placing and routing. Although quite satisfactory, we believe that more accurate results can be achieved by tuning more accurately each analytical FU model. The FU models used for the experiments have relative errors of up to 10%. For the presented benchmarks, the average relative errors are 8.14% and 1.9% for the estimation of LUTs and FFs, respectively. This is an important achievement since they are obtained without the binary optimizations performed by low level logic synthesis.

Table 1. Characteristics of the examples.

Example	# Loops	# Arrays	# JVM instr.	# basic blocks	# FUs	# loVHDL (data-path/FSM)
Fdct	3	3	625	10	215	2,166/589
Fdct: Part1	2	2	303	7	101	1,295/320
Fdct: Part2	1	2	335	4	111	1,354/313
Haar	8	4	201	25	113	1,415/464
Haar: Part1	2	2	73	7	42	831/176
Haar: Part2	2	2	29	7	15	621/117
Haar: Part3	2	1	73	7	41	823/176
Haar: Part4	2	1	29	7	15	621/117
FIR+Sobel	6	4	254	25	137	1,517/435
FIR	4	3	73	13	38	795/193
Sobel	2	2	178	13	99	1,220/281

Figure 5 shows the resources estimated for the FSMs of the examples. With one-hot encoding the number of FFs estimated corresponds to the correct number of FFs used by each design. With respect to the number of LUTs, the estimation has an average relative error of 6.8% for the considered benchmarks. Although this result is quite acceptable, we have plans to evaluate with more examples the LUTs needed to implement the FSMs generated by the compiler.

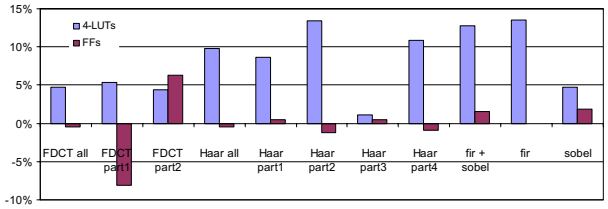


Figure 4. Ratio between the number of resources obtained by estimation and after logic and physical synthesis (ISE).

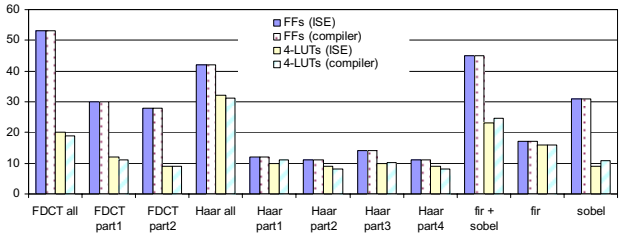


Figure 5. Number of resources (FFs and LUTs) obtained by estimation and after logic and physical synthesis (ISE) for the FSMs of the examples.

Table 2 presents results considering different implementations for each benchmark (with different number of configurations). We can see that most of times by splitting the original benchmark in more than one configuration, the maximum clock frequency increases (see last column of Table 2). Although the increasing is not high for all the examples, we believe that a higher performance impact may be achieved with larger examples. Work in progress is experimentally evaluating more complex benchmarks.

Table 2. Results obtained after logic and physical synthesis with ISE.

Example	# 4-LUTs	# Slices	# FFs	# Block RAMs	# MULT 18x18	freq. (MHz)	freq. increase (%)
Fdct	3,963	2,088	701	3	56	26.19	
Fdct: Part1	1,629	889	349	2	28	27.53	4.9%
Fdct: Part2	2,073	1,087	326	2	28	26.19	0.0%
Haar	1,464	899	598	4	6	67.71	
Haar: Part1	471	292	203	2	1	74.32	8.9%
Haar: Part2	238	133	85	2	2	93.36	27.5%
Haar: Part3	489	301	205	2	1	71.61	5.4%
Haar: Part4	237	146	108	2	2	100.89	32.9%
FIR+Sobel	1,524	1,005	646	4	6	55.41	
FIR	485	288	183	3	5	61.87	10.5%
Sobel	1,128	747	463	2	1	55.41	0.0%

5. Comments and Discussion

Compiling software programs that require large designs in FPGA devices, imposes a complex problem related to the loss of information between the programming statements existent in the source code and the computational structures created to form the final design. This gap is even more complex when advanced optimization techniques are used. This has been one of the main believed reasons why most hardware designers do not trust in high-level synthesis tools and may have been one of the main reasons for the modest success of those tools. That gap leads to a time-consuming task when the designer needs to control parts of the generated representation (in, e.g., an HDL), e.g., to specify timing constraints. Since most variables in the input program are promoted to wires, there are no association between those variables and registers in the final architecture[3]. Hence, accurate estimations must be researched in order to automatically perform most of the required tasks to achieve high-performance implementations. This is especially needed if we are trying to bring reconfigurable hardware to software programmers.

The high percentage of delays related to interconnections, which tends to be even more dominant, makes the estimation process very difficult and cumbersome. The results presented in this paper give first evidences that large designs obtained by compiling high-level computational descriptions defect from the long interconnections and thus make a case for temporal partitioning. However, to have temporal partitioning effective and advantageous to be massively used, FPGA companies need to invest on more advanced partial and dynamic reconfigurable schemes.

The presented results for estimation of resources at high abstraction levels show acceptable accuracy. However, we plan to tune each FU analytical model in order to quantify the achieved accuracy when complex and more accurate models are used for each FU. Estimation of the maximum clock frequency is a challenge very difficult to achieve with required accuracy at high abstraction levels. However, it is one of the most required issues when compiling to FPGAs and thus is worth research efforts.

Taking advantage of the resources of the recent FPGAs is not an easy task. The distributed RAMs, the block RAMs, and the distributed multipliers available on-the chip need more powerful binding schemes, able to make the best decision. Behavioral synthesis relies on complex resource-sharing schemes [14]. Many efforts have been done to achieve near-optimal resource sharing based on area or latency constraints. The efforts are easier to understand since the importance of the reduction of area to achieve competitive ASICs. However, when targeting FPGAs, the resources are already on-chip and one possible decision can be to select smaller or larger FPGA devices for a certain system. We believe that resource sharing in FPGAs only is worth to be applied for a set of operations (such as functions) and not for simple operations (e.g., additions).

6. Related Work

There is an extensive list of approaches aiming compilation from software programming languages to FPGA-based platforms [15][16][17][3]. The work with more affinities to our work is the compiler presented in [18][19]. They also aim the generation of specific architectures from a Java bytecode representation of the input algorithm[4].

Closely related to our work, are also research endeavors looking for better compiler optimizations able to take advantage of the FPGA features and estimation methods to assist compiler decisions.

Recent compiler approaches, extensively using scalar replacement, aiming a better usage of some of the particular FPGA resources, such as registers and memories, to cache data loaded from external memories [4] are an example of the former.

Estimation of resources and delays has been focused by a number of authors. Xu and Kurdahi [20] have been one of the first authors to enter into account wire delays in the timing estimation at RTL. Although high accuracy has been presented when targeting Xilinx XC4000 FPGAs, there is no evidence that the model works for the complex-

[3] This is also a problem that we face each time we need to debug the results produced by the compiler.

[4] Notice that there have been several efforts aiming the implementation of the JVM (Java Virtual Machine) in FPGAs.

ity of today's FPGAs. Furthermore, the approach works at logic levels to predict the mapping, shape and placement which may be not desired when compiling software programming languages. More related to our work is the approach presented in [21], where analytical models are used to estimate the resources needed to implement SystemC behavioral descriptions in a VirtexII-Pro. The authors use equations obtained by statistical analysis for calculating the number of LUTs and FFs.

Although with similarities to a number of research works, our approach differs from most of them by supporting complex analytical models for delay and resource estimations that can take advantage of the different bitwidths of the inputs and outputs of each operation. Our work is also to the best of our knowledge the first to show the impact on clock frequency of having several designs instead of a large design in the context of temporal partitioning.

7. Conclusions

This paper describes new developments through a compiler for FPGA-based reconfigurable platforms. The compiler is based on our previous efforts to compile Java bytecodes to FPGA-based platforms. We have shown how estimations are being carried-on in order to assist the scheduling and temporal partitioning tasks. The compiler accepts complex analytical models for estimation of delays and resources. The proposed estimation scheme shows quite promising results.

We show that smaller designs, achieved by temporal partitioning, may be best implementations than the use of a larger design in a single configuration. Although requiring more experiments, ultimately, this may impel FPGA companies to improve the support to dynamic and partial reconfiguration in the future FPGAs.

References

[1] R. Lysecky, and F. Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," in *Proc. Design Automation and Test in Europe Conference (DATE'05)*, Vol. 1, March 7-11, 2005, pp. 18-23.

[2] K. Compton, and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," in *ACM Computing Surveys*, Vol. 34, No. 2, June 2002, pp. 171-210.

[3] J. Cardoso, P. Diniz, and M. Weinhardt, "Compilation for Reconfigurable Computing Platforms: Comments on Techniques and Current Status," *INESC-ID Technical Report*, RT/009/2003, Lisbon, Portugal, Oct. 2003 (submitted to ACM Computing Surveys).

[4] N. Baradaran, J. Park, and P. Diniz, "Compiler Reuse Analysis for the Mapping of Data in FPGAs with RAM Blocks," In *Proc. IEEE Int'l Conference on Field-Programmable Technology (FPT'04)*, 2004.

[5] E. Panainte, K. Bertels, and S. Vassiliadis, "Instruction Scheduling for Dynamic Hardware Configurations," in *Proc. Design Automation and Test in Europe Conference (DATE'05)*, Vol. 1, March 7 - 11, 2005, pp. 100-105.

[6] J. Cardoso, and M. Weinhardt, "From C Programs to the Configure-Execute Model," in *Proc. Design, Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 3-7, 2003, pp. 576-581.

[7] S. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," in *Proc. Design, Automation and Test in Europe (DATE'05)*, 2005, pp. 66-67.

[8] J. Cardoso, and H. Neto, "Macro-Based Hardware Compilation of Java™ Bytecodes into a Dynamic Reconfigurable Computing System," In *Proc. IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, Napa Valley, CA, USA, April 21-23, 1999, pp. 2-11.

[9] J. Cardoso, and H. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," in *IEEE Design & Test of Computers Magazine*, March/April, 2003, vol. 20, no.2, pp. 65-75.

[10] J. Cardoso, "On Combining Temporal Partitioning and Sharing of Functional Units in Compilation for Reconfigurable Architectures," in *IEEE Transactions on Computers*, Vol. 52, No. 10, October 2003, pp. 1362-1375.

[11] R. Rodrigues, and J. Cardoso, "A Test Infrastructure for Compilers Targeting FPGAs," in *Proc. Int'l Workshop on Applied Reconfigurable Computing (ARC'05)*, Algarve, Portugal, February 22-23, 2005, IADIS Press, pp. 168-175.

[12] J. Cardoso, and P. Diniz, "Modeling Loop Unrolling: Approaches and Open Issues," in *Proc. of Int'l Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS IV)*, Samos, Greece, July 19-21, 2004, LNCS 3133, Springer Verlag, pp. 224-233.

[13] A. Nayak, et al., "Accurate Area and Delay Estimators for FPGAs," in *Proc. Design Automation and Test in Europe (DATE'02)*, March 2002, Paris, France, pp. 862-869.

[14] D. Gajski, *High-level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publisher, Boston, 1992.

[15] W. Böhm, et al., "Mapping a Single Assignment Programming Language to Reconfigurable Systems," in *The Journal of Supercomputing*, Kluwer Academic Publishers, vol. 21, no. 2, Feb. 2002, pp. 117-130.

[16] B. So, M. Hall, and P. Diniz, "A Compiler Approach to Fast Hardware Design Space Exploration for FPGA Systems," In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI'02)*, Berlin, Germany, June 17-19, 2002.

[17] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from Matlab™," In *Proc. IEEE/ACM Int'l Conference on Computer Aided Design (ICCAD'01)*, San Jose, CA, USA, November 4-8, 2001, pp. 314-319.

[18] M. Wirthlin , B. Hutchings , and C. Worth, "Synthesizing RTL Hardware from Java Byte Codes," in *Proc. 11th Int'l Conference on Field-Programmable Logic and Applications (FPL'01)*, 2001, pp. 123-132.

[19] J. Tripp, P. Jackson, and B. Hutchings, "Sea Cucumber: A Synthesizing Compiler for FPGAs," In *Proc. 12th Int'l Conference on Field-Programmable Logic and Applications (FPL'02)*, 2002, pp. 875-885.

[20] M. Xu, and F. Kurdahi, "Area and timing estimation for lookup table based FPGAs," in *Proc. European Design and Test Conference (ED&TC'96)*, March 11-14, Paris, France, 1996, pp. 151-157.

[21] C. Brandolese, W. Fornaciari, and F. Salice, "An Area Estimation Methodology for FPGA Based Designs at SystemC-Level," in *Proc. ACM/IEEE 41st Design Automation Conference (DAC'04)*, San Diego, CA, USA, June 7-11, 2004. pp 129-132.