C-Based Design Flow: A Case Study on G.729A for Voice over Internet Protocol (VoIP)

Mehrdad Reshadi, Bita Gorjara, Daniel Gajski
Center for Embedded Computer Systems (CECS),
University of California Irvine, CA 92697, USA.
{reshadi, gorjiara, gajski}@cecs.uci.edu

Abstract

In this paper we present the design of a G.729a codec in a C-based design flow. The codec is used in VoIP applications for sending speech over internet protocol. We started from the standard reference C implementation and generated several customized designs using the NISCT C-to-RTL toolset. Our final designs could run at very low clock frequencies (11 MHz for the decoder and 30 MHz for the coder) while meeting the timing requirements of the standard. We present these designs and the corresponding C-based design flow in this paper.

Categories and Subject Descriptors
C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems
C.4 [Computer Systems Organization]: Performance of Systems - *Design studies*

General Terms
Design, Algorithms

Keywords
NISC, HLS, ASIP, VoIP, G729A, C-to-RTL, C-based design flow

1. Introduction

Algorithm-level synthesis and C-based design flow enable design at higher-level of abstraction resulting in higher productivity and shorter design times. Typically a C-description can be turned into RTL implementation using either Application Specific Instruction-set Processors (ASIP)[1] or High-Level-Synthesis (HLS)[2]. In ASIP, such as Tensilica [3], LISA [4][5], and EXPRESSION [6]; a base processor is extended by adding custom instructions that can improve the performance of a specific application. Such approaches usually support the complete C grammar and are useful for large applications. On the other hand, in HLS, the C-description is directly translated to a micro-architecture (datapath and controller) that executes only the given application. Such approaches usually support a sub-set of the C grammar and are useful for small applications. Many variations of ASIP and HLS have been studied by researchers or offered as commercial products.

In this paper, we show the results of designing a complete G.729 algorithm which is used in Voice over Internet Protocol (VoIP) applications. In a VoIP application, analog voice signals such as telephone calls and faxes are converted to digital packets and then transmitted over the network. VoIP applications use speech compression algorithms to reduce the bandwidth requirements. G.729 codec is a popular speech compression algorithm based on Code-Excited Linear Prediction [7] compression. In G.729 the voice is coded into 8-kbps streams. The coder/decoder work on 10ms frames and can have up to 15ms delays. The G.729 codec is standardized by International Telecommunication Union (ITU-T)[8] and has several variations. The G.729 Annex A (G.729a) is a more popular version that is compatible with G.729 but has a lower computational complexity with lower but still acceptable quality. The ITU provides C code of the reference implementation of the G.729a. Our goal is to start from this C code and explore different options to achieve the best design. We user the No-Instruction-Set-Computer (NISC) Technology toolset [9] because the same toolset can be used for both ASIP design as well as custom design generation similar to HLS. Currently, no other one toolset provides both ASIP and HLS approach. Also, typically these tools require significant changes in the C code which would require significant time and cancels out the productivity gains of using a C-based design flow. Using NISC toolset, we implemented the algorithm using (1) a general purpose processor, (2) an automatically generated custom datapath using general operations (HLS), (3) a general purpose processor extended with a custom functional unit (ASIP), and (4) an automatically generated datapath using general operations as well as a custom functional unit (ASIP+HLS). Depending on the area and performance requirements one can choose any of the designs. The highest performance was achieved when we combined both ASIP and HLS to generate a fully customized architecture for each of the applications. The final design could run at 11 MHz for the decoder and 30 MHz for the coder while meeting the timing requirements of the standard.

The rest of this paper is organized as follows: Section 2 explains the NISC design flow that we used in our experiments. Section 3 describes how several different designs were generated for the G.729a speech compression algorithm and Section 4 summarizes the problems we faced in the overall flow that can be improved in future. Finally, Section 5 concludes the paper.

2. NISC Design Flow

In NISC design approach, the target architecture is a variation of microcoded architectures [10][11] and is composed of a datapath and a controller. The datapath of NISC can be simple or as complex as datapath of a processor. The controller drives the control signals of the datapath components in each clock cycle. These control values are generated by the NISC cycle-accurate compiler [12]. These values are either stored in a memory or generated via logic in the controller. Both the controller and the datapath can be pipelined. Figure 1 shows a sample NISC architecture with a pipelined datapath that has partial data forwarding, multi-cycle and pipelined units, as well as data memory and register file.

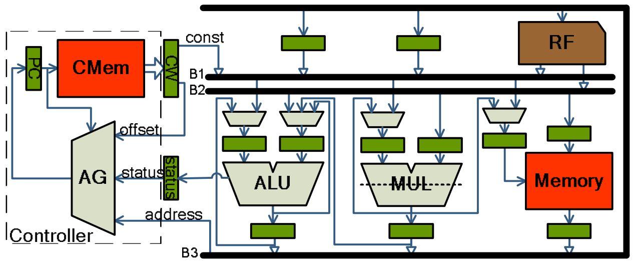

Figure 1. NISC architecture example

Figure 2 shows the design flow for designing a custom NISC for a given application. In NISC, the controller is generated after compiling the application on a given datapath. Therefore both the application and the datapath description are considered input to the NISC cycle-accurate compiler. The datapath can be directly described or automatically generated based on the application behavior. The datapath is captured in the GNR (Generic Netlist Representation) format [13] which describes the datapath as a netlist of components and assigns different attributes to each component. A component in datapath can be a register, register-file, bus, multiplexer, functional unit, memory etc. The functionalities of components are associated with timing information of corresponding control values.

The GNR description of the datapath and the C code of the application are then given as input to the NISC compiler. The NISC compiler maps the application directly on the given datapath and generates a Finite State Machine (FSM) that determines the behavior of datapath in each clock cycle. Finally, the complier generates the contents of data memory (if any) and also uses the FSM to generate the stream of control values. After applying several controller/FSM optimizations [14], the RTL generator, first synthesizes a controller from the output of compiler, and then uses the datapath information to generate the final synthesizable RTL design (described in Verilog). This RTL is then used for simulation (validation) and synthesis (implementation). After synthesis and Placement and Routing (PAR), the accurate timing, power, and area information can be extracted form the layout and used for further datapath *refinement*. For example, the user may add functional units and pipeline registers, or change the bit-width of the components and observe the effect of modifications on precision of the computation, number of cycles, clock period, power, and area. The NISC toolset also includes an automatic datapath generator/refiner [15]. In NISC, there is no need to design the instruction-set because the compiler automatically analyzes the datapath and extracts possible operations and branch delay. Therefore, the designer can refine the design very fast.

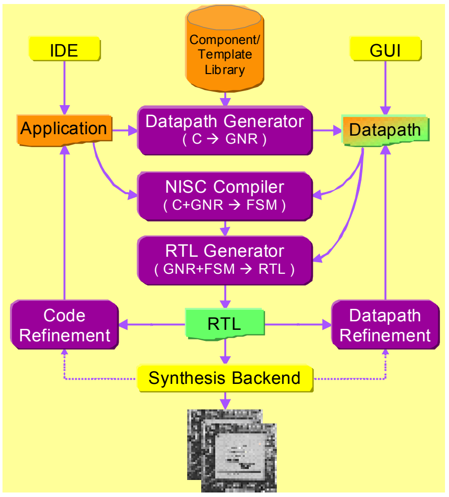
IDE
Component/ Template Library
GUI
Application
Datapath Generator (C → GNR)
Datapath
NISC Compiler (C+GNR → FSM)
RTL Generator (GNR+FSM → RTL)
Code Refinement
RTL
Datapath Refinement
Synthesis Backend

Figure 2. NISC design flow

Similar to NISC, macrocode architecture has also been the target of other design approaches such PICO [16][17], ARM OptimoDE [18][19], MIMOLA [20], and TIPI [21]. However, in these approaches, either the architecture is fixed and cannot be customized, or a complete toolset is not available.

3. Design of G729 for VoIP

3.1 Preparing the source code

We downloaded the C code of the reference implementation from ITU website [8]. The code contains two separate projects for the coder and the decoder of the G.729a. Excluding comments and blank lines, the coder contains about 7000 lines of code and the decoder contains about 5500 lines of C code. Parts of the code are shared between the coder and decoder. This reference code was written for execution on a desktop PC. In the code, it was assumed that the input is read from a file and the output is finally written to another file. Also, in several parts of the code, the status of the application was printed on the console (using `printf` function). Clearly, this kind of IO is not efficient or even possible in an embedded implementation. So, the first step of the preparation of the code was to remove all references to all functions of the C standard IO library (i.e. `stdio.h`). These are the only changes needed in the C code since NISC toolset supports full ANSI C syntax. For reading the input of coder/decoder and writing the output, we assumed a structure as shown in Figure 3.

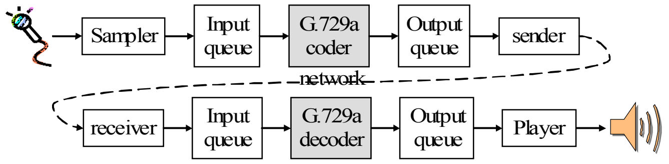

Figure 3. Block diagram of VoIP application

We updated the `main` function of the coder and the decoder to read from the input queue and write the data to the output queue. Figure 4 shows the updated C code. The NISC toolset uses a feature called *pre-binding* which assigns a C function to the low-level operations implemented in hardware [22]. These functions are described in the GNR description of the components. For example, a queue component implements `push()`, `pop()`, and `size()` functions for adding values to the queue, removing values from it, or reading the number of available values in the queue, respectively. The queue implementation is captured in GNR. GNR supports both RTL and external IP (e.g. dedicated components in FPGA) implementation for any component. When we add a queue to the datapath, the NISC compiler extract the pre-bound functions based on the instance name of the queue. For example, for queue `input_queue`, pre-bound functions `__$input_queue_push()`, `__$input_queue_pop()`, `__$input_queue_size()` can be used in the C code to directly access the functionalities of the queue. The C code in Figure 4 first waits for the previous phase (*sampler* for coder and *receiver* for decoder in Figure 3) to insert all values of the frame in the input queue. After processing the frame, it first makes sure that the next phase (*sender* for coder and *player* for decoder in Figure 3) has already consumed the previous frame and then write the processed results of the current frame. In this way, all blocks of Figure 3 can work in parallel in a pipelined fashion.

```
void main() {
  extern Word16 *input; //Pointer to input
  extern Word16 *output; //Pointer to output
  //...
  while(1) {
    while(__$input_queue_size() < FRAME_SIZE);
    for(i=0; i<FRAME_SIZE; ++i)
      input[i] = __$input_queue_pop();
    process_frame(); //encode or decode the frame
    while(__$output_queue_size() != 0);
    for(i=0; i<FRAME_SIZE; ++i)
      __$output_queue_push(output[i]);
  }
}
```

Figure 4. Main body of the coder/decoder

3.2 The target architecture

An important and useful feature of the NISC design flow (Figure 2) is that we can manually describe the datapath and have the toolset to behave as an ASIP tool, or we can let the toolset automatically generate the datapath and hence act as an HLS tool. In this way, we did not need to learn two sets of tools and prepare two different versions of the C code once for an ASIP tool and once for an HLS tool. In other words, we can easily explore a full range of general purpose to fully customized architectures with NISC toolset. In this section, we demonstrate this capability.

We started from a general-purpose architecture shown in Figure 5. We first described the datapath of this architecture in GNR and then used the NISC toolset to compile the application on this architecture and generate the RTL. We synthesized this RTL on a Xilinx Virtex4 (90-nm) FPGA package. The second rows of Table 1 and Table 2 show the result for coder and decoder, respectively. In these tables, the second columns show the number cycles it takes to process one frame.

The third columns show the achieved frequency after synthesis and the fourth columns show the minimum required frequency for processing the frame in less than 15ms (required by standard). The fifth columns show the size of microcode and the sixth columns show the size of data memory. Finally, the last two columns show the number of FPGA slices and RAM blocks used in each design. From these tables, it is clear that, with the general purpose datapath of Figure 5, we can meet the deadline for G.729a decoder, but not for the G.729a coder.

In the next step, we use the NISC toolset as an HLS tool and generate a custom datapath for each of the G.729a coder and the G.729a decoder. For each application, the toolset iterates several times (according to the flow of Figure 2) to generate a custom datapath and refine it until a good performance improvement and resource utilization is achieved. In each iteration, the tools explore different datapath netlists by changing number of functional units of each type and their interconnects [15]. The third rows of Table 1 and Table 2 show the custom datapath (CDP) generation results for coder and decoder, respectively. Note that after this customization step, the number of cycles per frame, and the microcode size of both applications are reduced and the area (number of slices) is increased. However, still after this customization, the design of the coder cannot meet the timing requirements.

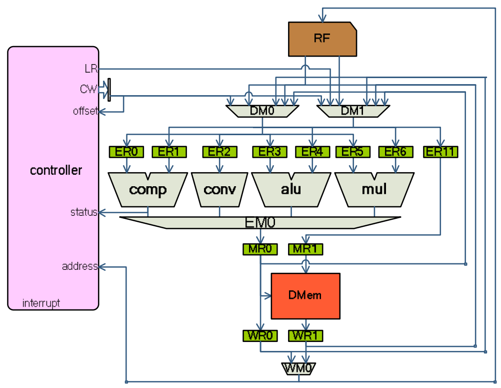

Figure 5. General purpose datapath (GDP)

It may be possible to further improve the performance by breaking the application into several parallel blocks and then use the HLS mode of the NISC toolset for each block. However, as we mentioned before, the reference code is written assuming a sequential execution on a desktop PC. Therefore, partitioning the code into parallel blocks requires significant code modifications. Our goal was to see if we can meet the design constraints with minimum amount of code modifications. Therefore, instead of partitioning the code, we tried an ASIP approach and evaluated the improvements.

To use an ASIP approach we need to find proper parts of the code that can improve the overall application performance if we run those parts on a custom functional unit. We profiled the execution of the application on GDP and realized that in both coder and decoder, almost half of the execution time was spent in only two functions, i.e. L_mac() and L_msu(). These two functions in turn call L_mult(), L_add(), and L_sub(). The latter three functions perform a saturated version of the multiply, add, and subtract operations on 16-bit integers. These functions are also used in other parts of the code as well. We noticed that implementing the latter three functions in hardware is a lot easier and more efficient than directly implementing L_mac() and L_msu() functions in hardware. Implementing L_mac() and L_msu() functions in hardware can also significantly increase the critical path delay and hence reduce the maximum possible clock frequency for the design. Additionally, implementing the three saturated operations in hardware can improve the performance of other parts of the code as well. We decided to add a custom functional unit that performs the three saturated operations L_mult, L_add, and L_sub. Before moving these functions to hardware, we have to make sure their global side effects (if any) are properly handled. In the reference code, global variable Overflow was accessed or updated in many places including the above functions. Since these functions also need to access the overflow flag, our custom functional unit also contains a register that is updated / accessed by its own operations, or can be updated/accessed directly in the program. Figure 6 shows the block diagram of general datapath of Figure 5 which now has an extra custom functional unit, i.e. *cfu*. After analyzing the GNR description of Figure 6, the NISC toolset generates five pre-bound functions that can be used in the C code to directly access the custom functional units. These functions include: __$cfu_L_mult(), _$cfu_L_add(), __$cfu_L_sub(), __$cfu_getOverflow(), __$cfu_setOverflow(). These functions are in fact equivalent to custom instructions in an ASIP approach. To use the new custom functional unit, we need to replace all calls to L_mult(), L_add(), and L_sub() in the code with __$cfu_L_mult(), __$cfu_L_add(),and __$cfu_L_sub() pre-bound functions. Also, we must replace all reads from Overflow variable with __$cfu_getOverflow(), and all writes to Overflow variable with __$cfu_setOverflow(). These changes were relatively straight forward and we used C macros to minimize the necessary changes in the code.

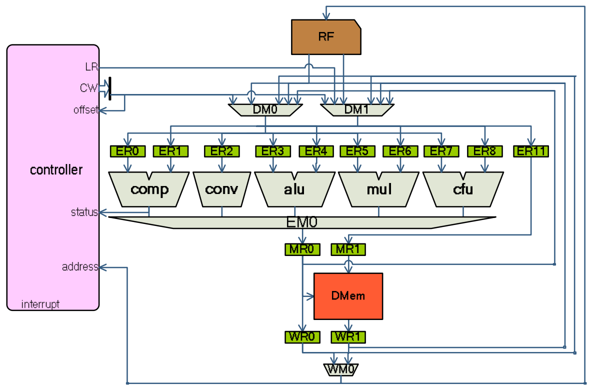

Figure 6. General purpose datapath + custom function unit (GDP+CFU)

The fourth rows of Table 1 and Table 2 shows the results of running the modified coder/decoder on the GDP+CFU (Figure 6). In this case, the required clock frequency for processing one frame under 15ms is much lower. Also, compared to the general purpose datapath (GDP), adding the custom functional unit to GDP+CFU reduces the microcode size and increases the area; which is expected in a typical ASIP approach.

Table 1. Results for G.729a coder

Arch	#cycles / frame	Freq (MHz)	Min Req. Freq (MHz)	µcode size(KB)	DMem size (KB)	#slices	#RAM Blocks
GDP	2664928	126	178	82	7.5	1619	48
CDP	1935112	119	129	74	7.5	2016	47
GDP+CFU	642848	112	43	75	7.5	1760	45
CDP+CFU	452886	91	30	70	7.5	2418	44

Table 2. Results for G.729a decoder

Arch	#cycles / frame	Freq (MHz)	Min Req. Freq (MHz)	µcode size(KB)	DMem size (KB)	#slices	#RAM Blocks
GDP	571780	126	38	33	7.2	1092	27
CDP	413835	111	28	31	7.2	1588	27
GDP+CFU	289150	112	19	31	7.2	1200	26
CDP+CFU	166625	87	11	31	7.2	1950	26

To maximize the improvements, we combined the ASIP and HLS modes of the NISC toolset. We added our custom functional unit to the list of available resources and ran the tools to generate a custom datapath for each of the coder and decoder applications. In this step, we used the version of coder and the decoder that contained the pre-bound functions for using the custom functional unit. The last rows of Table 1 and Table 2 show the results of generating custom datapath with custom functional unit (CDP+CFU). The performance is further improved and the required clock frequency is further decreased. Also, the microcode size is further decreased while the area is increased. The final design of the coder runs 4.2 times faster than the general purpose and the decoder runs 2.4 times faster when all run at their highest possible clock frequencies.

Depending on the actual application of the G.729a codec, one can choose any of the above designs. Also, we can run each design with the minimum required frequency and save power; or run them at them maximum possible frequency to process more speech frames.

4. Lessons learned

It is evident that using a C-based design flow can significantly reduce the design time while achieving the desired results. However, the success of the project, the quality of the final results, and the benefits of a c-based design flow depend directly on (i) the quality of the input C code, (ii) quality of high-level C to RTL tool, and (iii) proper integration of the C to RTL tool with the backend synthesis tools.

To further increase the impact and benefits of C-based design flows, the software developer and providers of reference code for standards need to consider, almost from the beginning, that their code is not targeted only for execution on a general purpose processor and it can be used to directly generate RTL. Many researchers (e.g. [23]) have studied the effect of coding styles on the quality of synthesis from C. Availability of more synthesis friendly ANSI C/C++ code can significantly increase the acceptance of C-based design flows among designers.

On the other hand, C to RTL tool developers must also consider that supporting a bigger subset of standard C/C++ and avoiding proprietary extensions is crucial for success of C-based design. One of the major premises of using a C-based design is improved productivity. The productivity benefits can be reduced or even negated if significant code modifications in the applications are needed before a C to RTL tool can be used. We believe it is almost equally important to develop code refinement tools that are targeted to C to RTL design flow. These tools can further automate the tedious and error prone code modifications. General code refactoring tools [24][25] usually are helpful but may not directly target the specific needs of C-based hardware design flows. Although some researchers [26][27] have started looking at such issues, the design community can benefit from more similar research and eventually commercial products.

Today, almost all C-based design tools are offered separately from the backend synthesis tools (RTL synthesis, logic synthesis, physical synthesis, etc.). Many optimizations and customizations in high-level tools require feedback from the lower-level tools. Despite the apparent business incentives in controlling the market, the backend tool providers can themselves enjoy a much larger market if they consider and provide facilities for tighter integration of other third party tools in their flow. In the long run, initiatives such as SI^2 [28] can facilitate this integration. But until then, the backend tool vendors can significantly improve the situation by for example maintaining a consistent (and backward compatible) report generation scheme. For example we noticed that Xilinx ISE reports the number of RAM blocks as "Number of FIFO16/RAMB16s" for Virtex4 and "Number of RAM/FIFO" for Virtex5. Such simple changes in the reports of the backend tool can break the high-level tools that depend on them.

5. Conclusion

In this paper we studied the design of a G.729a speech compression codec from a reference C code implementation. We showed that it is possible to achieve efficient designs and meet the standard requirements with very little code modifications and automatic RTL generation using NISC toolset. We analyzed the results of using four approaches for implementing the algorithm. Namely, we tried using (1) a general purpose processor, (2) using HLS alone, (3) using ASIP alone, and (4) combining ASIP and HLS. The latter gave us the best results.

6. References

[1] M.K. Jain, M. Balakrishnan, A. Kumar, "ASIP Design Methodologies: Survey and Issues", In *Proceedings of the Fourteenth International Conference on VLSI Design*, 2001.

[2] D. Gajski, N. Dutt, A. Wu, S. Lin, "*High-Level Synthesis Introduction to Chip and System Design*", Kluwer Academic Publishers, The Netherlands, 1994.

[3] Tensilica Inc. http://www.tenisilica.com

[4] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, A.Wieferink, H. Meyr, "A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language", *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, 2001.

[5] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, A. Nohl, "RTL Processor Synthesis for Architecture Exploration and Implementation", *DATE*, 2004.

[6] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability", *DATE*, pages 485-490, 1999.

[7] M. Schroeder and B. Atal, "Code-excited linear prediction (celp): High-quality speech at very low bit rates", in *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing*, 1984.

[8] ITU-T Website: http://www.itu.int/

[9] NISC Technology website http://www.cecs.uci.edu/~nisc/.

[10] A. Agrawala, T. Rauscher, Foundations of Microprogramming: "*Architecture, Software, and Applications*", Academic Press, 1976.

[11] S. Habib, "*Microprogramming and Firmware Engineering Methods*", John Wiley & Sons, Inc., 1988.

[12] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", *CODES+ISSS*, 2005.

[13] B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs", in *International Conference on Computer Design (ICCD)*, 2006.

[14] B. Gorjiara, D. Gajski, "FPGA-friendly Code Compression Technique for Statically Scheduled Horizontal Microcoded Custom Ips", *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2007.

[15] B. Gorjiara, D. Gajski, "Automatic Architecture Refinement Techniques for Customizing Processing Elements", *DAC*, 2008.

[16] M. Sivaraman, S. Aditya, "Cycle-time aware architecture synthesis of custom hardware accelerators", in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2002.

[17] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist and M. Sivaraman, "PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators", in *Journal of VLSI Signal Processing*, 31(2), 2002.

[18] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, K. Van Nieuwenhove, "OptimoDE: Programmable Accelerator Engines Through Retargetable Customization", *Hot Chips*, 2004.

[19] M. Byatt, "Data plane processing with configurable architectures", *ARM white paper*, 2003.

[20] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer, "The MIMOLA Language - Version 4.1. Technical Report." Computer Science Dpt., University of Dortmund, Sept. 1994.

[21] S. Weber and K. Keutzer, "Using Minimal Minterms to Represent Programmability", *CODES+ISSS*, 2005.

[22] M. Reshadi, D. Gajski, "Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-Scheduled Horizontal-Microcoded Architectures in Embedded Systems", *DATE*, 2007.

[23] G. Stitt, F. Vahid, W. Najjar, "A Code Refinement Methodology for Performance-Improved Synthesis from C", *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006.

[24] M. Fowler, "Refactoring: Improving the design of existing code", in *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, Springer-Verlag, 2002.

[25] Refactoring catalog, http://www.refactoring.com/catalog/

[26] Pramod Chandraiah, Rainer Dömer, "Pointer Re-coding for Creating Definitive MPSoC Models", *CODES+ISSS*, 2007.

[27] Sumit Gupta, R.K. Gupta, N.D. Dutt, A. Nicolau, "*SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*", Kluwer Academic Publishers, 2004.

[28] Silicon Integration Initiative website: http://www.si2.org/