Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 685128, 13 pages
doi:10.1155/2008/685128

Research Article

A Case Study: Quantitative Evaluation of C-Based High-Level Synthesis Systems

Omar Hammami,[1] Zhoukun Wang,[1] Virginie Fresse,[2] and Dominique Houzet[3]

[1] *EECS Department, National Higher School of Advanced Techniques (ENSTA), 32 Boulevard Victor, 75739 Paris, France*
[2] *Hubert Curien Laboratoire (UMR CNRS 5516), Université Jean Monnet, 18 Rue Benoit Lauras, 42000 Saint Etienne, France*
[3] *Laboratoire Grenoble Images Paroles Signal Automatique (GIPSA Laboratoire) (UMR CNRS 5216), Institut National Polytechnique de Grenoble (INPG), 46 Avenue Félix Viallet, 38031 Grenoble, France*

Correspondence should be addressed to Omar Hammami, profhammami@yahoo.com

Received 5 October 2007; Revised 25 February 2008; Accepted 14 May 2008

Recommended by Sandeep Shukla

Embedded systems require a high-level abstraction modeling to tackle their complexity and improve design productivity. C-based design techniques and methodologies have been proposed for this purpose. Various IPs with diverse complexities and functionalities can be selected to build a system. However, area and energy issues are paramount in this selection process. Implementation hints should therefore be available at the highest possible level of abstraction. Some C-based tools have been proposed in this order through either true synthesis or estimation. In this paper, we conduct through a case study an evaluation of C-based design of embedded systems and point out the impact of behavioral synthesis on embedded systems multiobjective high-level partitioning.

1. INTRODUCTION

Embedded systems are increasingly complex to design, validate, and implement [1]. System composition of embedded processors and hardware accelerators leads to the HW/SW codesign methodologies. Traditional codesign methodologies require that hardware and software are specified and designed independently. Hardware and software validations are thus done separately without interface validations. The partitioning is therefore decided in advance and as changes to the partition can necessitate extensive redesign elsewhere in the system, this decision is adhered as much as possible. It can lead to suboptimal designs as partitioning techniques often rely on the designer's experience. This lack of a unified hardware-software representation can lead to difficulties in verification of the entire system, and hence to incompatibilities across the hardware/software boundary. Using a single language for both simplifies the migration task and ensures an entire system verification. Important uses of a unified design language in addition to synthesis are validation and algorithm exploration (including an efficient partitioning). C-like languages are much more compelling for these tasks, and one, in particular SystemC [2], is now widely used for system-level design [3–5], as are many ad hoc variants. Rapid C synthesis allows also fast emulation techniques of hardware and software used for architecture exploration. C-based synthesis fundamentals emerged from these HW/SW codesign methodologies with the objective of reducing the design productivity challenge [1]. However, although raising the system design level of abstraction contributes to reduce the design complexity, reliable and predictable silicon implementation remains mandatory which supports high-level design handoff [1]. The question is then on the impact of C-based behavioral synthesis on C-based specification and modeling framework for embedded systems.

This work evaluates through a case study the performance and efficiency of several high-level description languages (SystemC, Handel-C) for FPGA-based embedded systems. The rest of the paper is organized as follows: Section 2 presents the related work while Section 3 reviews main issues regarding C-based synthesis system design methodology. Design case studies along with performance evaluation studies and results are fully described in Section 4. Finally, Section 5 presents the conclusions.

2. RELATED WORK

Intensive research has been conducted on C-based behavioral synthesis. With the evolution of system-level design languages, the interest in efficient hardware synthesis based on behavioral description of a hardware module has also been visible. For a system designer, the behavioral synthesis is very attractive for hardware modeling as it leads to significant productivity improvements. Important work has been done in academia on behavioral synthesis with C/SystemC [6–14]. For example, the work in [6] has presented a synthesis environment and the corresponding synthesis methodology. It is based on traditional compiler generation techniques, which incorporate SystemC, VHDL, and Verilog to transform existing algorithmic software models into hardware system implementations. In [12], the authors address the problem of deriving exact finite state machines (FSMs) from SystemC descriptions which is the first part of behavioral synthesis methodologies. In an effort to extend synthesis to object-oriented constructs of C++ language, [10] presents an approach to object-oriented hardware design and synthesis based on SystemC. Behavior synthesis problem is multiobjective in nature where synthesis tools allow the hardware designers to customize the behavioral synthesis process through various options which constitute a huge design space. In this situation, solution exists of a set of optimized results represented in the form of Pareto curves and Pareto surfaces. In [14], a methodology that allows the designers to generate and analyze the best synthesis results based on area, performance, and power consumption estimation through an automatic exploration of synthesis results is presented. ROCCC [15] and Spark [11] are C-to-VHDL high-level synthesis academic frameworks.

Some tools for behavioral SystemC/C synthesis are available in the market. Synopsys SystemC compiler [16] was perhaps the first commercial effort to synthesize behavioral code written in ESL languages. Celoxica's agility [17] and Forte Design [18] can synthesize a SystemC behavioral description of hardware modules. They also give the area estimation requirements for various ASIC and FPGA-based architectures. Orinoco Dale [19], ImpulseC [20], and CatapultC [21] estimate the area and the energy for a C description of an application.

C-based behavioral synthesis can be used at system-level design in order to guide system partitioning. Reference [4] presents a framework for the generation of embedded hardware/software from SystemC. In [5], area and energy are estimated for various IP components in the system before actually modeling the system in TLM. The area and energy estimation information is fed into the TLM model of the system where we partition the system by automatically exploring the system design space based on the given information. This design flow uses the TLM modeling for system design space exploration. It includes area and energy estimation information during the partitioning process. This work represents a good layout foundation for addressing the impact of implementation on embedded systems.To the best of our knowledge, our study is the first attempt to the contribution of benchmarking these tools for performance comparison purposes and to analyze the interaction with place and route tools options.

3. C-BASED SYNTHESIS

3.1. C-language fundamentals

Some C-language characteristics are troubling when synthesizing hardware from C. Edwards listed in [22] the key challenges of a successful hardware specification language: concurrency model, types, timing specification, memory and communication patterns, hints, and constraints. The C-language has no support for parallelism. Either the synthesis tool is responsible for finding it or the designer must modify the algorithm by inserting explicit parallelism. C-hardware language designers adopted different parallelism strategies. Communication patterns depend on the parallel programing model provided by the C-hardware languages. These communication patterns do not exist in C-language. The C-language is also mute on the subject of time. It guarantees causality among most sequences of statements but says nothing about the amount of time it takes to execute each. It is essential to find reasonable techniques for specifying hardware needs and mechanisms for specifying and achieving timing constraints. Data types are another major difference between hardware and software languages. The most fundamental type in hardware is a single bit traveling through a memoryless wire. Variable width integer is natural in hardware yet C does not support variable width. C's memory model is a large undifferentiated array of bytes, yet many small varied memories are most effective in hardware. All these characteristics must be considered when designing C-like hardware languages. All characteristics related to the considered tools are analyzed in the rest of the paper.

3.2. HLS approaches and tools

Various C-based hardware description languages have been proposed over the past decade. These tools use different approaches for timing, parallelism, data types, and communication modeling. These approaches can be either automatic or manual.

(i) Concurrency model. Coarse grain and fine grain parallelism are available for most of the approaches. The communication between tasks is the coarse grain parallelism. The fine grain parallelism is the operator parallelism and pipeline. This parallelism is either explicit or implicit. Constructs dedicated to the parallelism are added to the C-language for the explicit parallelism programing. Additional constructs are not required for the implicit parallelism. The level of parallelism can be handled with constraints and compile options.

(ii) Types. All approaches ensure hardware bit-true data type manipulation and declaration with additional data types. The size of the data can be precisely controlled for each operating stage.

Table 1: Approaches used for C-based hardware description languages.

Features	Approaches	Language	Tools
Concurrency model	Implicit	C	Spark, Impulse CoDeveloper, CatapultC, ROCCC,
	Explicit	Handel-C, SystemC	DK Design Suite, Forte Cynthesizer, and agility
Types	Explicit	All languages	All tools
Timing specification	Implicit	C	Spark, Impulse CoDeveloper, CapatultC, ROCCC,
	Explicit	Handel-C, SystemC	DK Design Suite, Forte Cynthesizer, and agility
Memory	Implicit	C	Impulse CoDeveloper
	Explicit	Handel-C, C	DK Design Suite
Communication patterns	Implicit	C	Spark
	Explicit	Handel-C	DK Design Suite, Impulse CoDeveloper, and agility

Table 2: Case study selected C-based environments.

Tool	Language	Company
Impulse CoDeveloper	ImpulseC	Impulse Accelerated Technologies
DK suite	Handel-C	Celoxica
Agility SystemC compiler	SystemC	Celoxica

(iii) Timing specification. All approaches take the latency and the throughput into account. The approaches are either explicit or implicit. Each clock cycle is precisely described for the explicit approach. The timing constraint only concerns the clock period. Tools with an implicit timing approach generate the scheduling and the parallelism of operators to meet the latency and the throughput timing constraints.

(iv) Memory. Internal memories are either RAM or registers. They can be either implicitly selected or explicitly specified.

(v) Communication patterns. Predefined communication and memory protocols are used for the implicit approach. For the explicit approach, the protocols are described in details in the code with dedicated instructions.

The chosen approach can lead to substantial code modifications.

Each tool uses specific approaches for concurrency model, types, timing specification, and memory and communication pattern. Several tools cannot be studied in this paper but they can be classified in the following table (see Table 1).

Several tools can be evaluated with the presented approaches. Three selected tools given in Table 2 are selected on the basis of our own design experience and the used approaches. It is indeed of importance that a significant amount of design experience is needed in order to compare the design approaches with the design. Each tool has different approaches for concurrency, data types, timing specifications, memory and communication.

3.2.1. ImpulseC

Impulse CoDeveloper is an ANSI C synthesizer [20] based on the ImpulseC language with function libraries supporting embedded processors and abstract software/hardware communication methods including streams, signals, and shared memories. This allows software programmers to make use of available hardware resources for coprocessing without writing low-level hardware descriptions. Software programmers can create a complete embedded system that takes advantage of the high-level features provided by an operating system while allowing the C programing of custom hardware accelerators. The ImpulseC tools automate the creation of required hardware-to-software interfaces, using available on-chip bus interconnections.

(i) *Concurrency model*. The main concurrency feature is pipelining. As pipelining is only available in inner loops, loop unrolling becomes the solution to obtain large pipelines. The parallelism is automatically extracted. Explicit multiprocess is also possible to manually describe the parallelism.

(ii) *Types*. ANSI C-type operators are available like “int” and float as well as hardware types like “int2,” “int4,” and “int8.” The float to fixed point translation is also available.

(iii) *Timing specification*. The only way to control the pipeline timings is through a constraint on the size of each stage of the pipeline. The number of stages of the pipeline and thus the throughput/latency are tightly controlled.

(iv) *Memory*. All arrays are stored either in RAM or in a set of registers according to a compilation option.

(v) *Communication patterns*. Streams (FIFO) with different formats are available as well as signals and shared memories interface.

3.2.2. DK design suite tool

DK Design Suite is a complete Electronic System Level (ESL) environment supporting the Handel-C language [23]. It provides the user with a complete flow: from specification to implementation such as architecture-optimized EDIF

Netlist for FPGA's RTL Verilog or VHDL used for alternative synthesis flows and other hardware targets including ASIC designs.

Celoxica's Handel-C is a language for digital logic design that has many similarities to ANSI-C. Handel-C is a variant that extends the language with constructs for parallel statements and OCCAM-like rendez-vous communication.

Handel-C language and the IDE tool introduced by Celoxica provide both simulation and synthesis capabilities.

(i) *Concurrency model*. The application is written with sequential programs in Handel-C. Programs written in Handel-C are implicitly sequential: writing one command after another indicates that those instructions should be executed in that exact order. Parallel constructs are possible with the *par* keyword to gain maximum benefit in performance from the target hardware.

(ii) *Types*. This language adopts the manual customization of numerous representations. Handel-C types are not limited to specific widths. Any defined Handel-C variable can be specified with the minimum width required to minimize hardware usage.

(iii) *Timing specification*. DK Design Suite includes a cycle accurate multithread symbolic debugger. Handel-C timing model is uniquely simple: any assignments or delay take one clock cycle.

(iv) *Memory*. Each data storage is explicitly specified in a RAM or a set of registers by the programmer.

(v) *Communication patterns*. Any external specification with any type of communication protocol can be described. Handel-C allows you to target components such as memory, ports, buses, and wires.

3.2.3. Agility compiler

The agility compiler from Celoxica is described below [17]. The agility compiler provides a behavioral design and synthesis for SystemC. It is a single solution for FPGA design and ASIC/SoC prototyping. Early SystemC models can be quickly realized in working silicon yielding accurate design metrics and RTL for physical design.

(i) *Concurrency model*. Explicit multiprocess is possible to manually manage parallelism. Each stage of the pipeline can be manually described with the use of wait() instructions in an RTL-like style.

(ii) *Types*. ANSI C types and operators such as "int" and "char" can be used. Agility compiler also accepts hardware types such as sc_uint⟨8⟩, sc_int⟨20⟩, and fixed point sc_fix⟨⟩.

(iii) *Timing specification*. The SystemC timing model is uniquely simple: all assignments between two wait() take one clock cycle. This modeling style is similar to the VHDL "wait until rising_edge(clk)" style.

(iv) *Memory*. SystemC provides supports for interfacing to on-chip RAMs and ROMs using dedicated array keywords. If not used, arrays are stored in a set of registers.

(v) *Communication patterns*. Any external specification with any type of communication protocol can be described.

4. DESIGN CASE STUDIES

In order to evaluate the synthesis efficiency of the previously described tools, the use of commonly accepted benchmarks for C-based synthesis would have been useful. However, so far no benchmarks have been released from the OSCI synthesis working group which defined the synthesizable subset of neither SystemC nor by any other body. Therefore, we decided to compose our own case studies which are basic and simple functions to ensure reproducibility.

4.1. Designs examples

Evaluation consists in studying the efficiency of the synthesis from C-based hardware descriptions. Common benchmarks are used for the evaluation of the previously described tools.

Our own case studies consist in a set of short and simple functions to allow reproducibility. The selected cases are two 3×3 image filters [24], the FFT, and an octree traversal algorithm (ray casting in projective geometry RCPG) [25].

4.1.1. Linear filter

A linear filter and a nonlinear filter are chosen. The two filtering benchmarks are based on a $3*3$ window core processing. The linear filter is the mean filter [24]. The mean filter is the simplest type of low-pass filter. The mean or average filter is used to soften an image by averaging surrounding pixel values in a 3×3 window. This filter is often used to smooth images prior to processing. It can be used to reduce pixel flicker due to overhead fluorescent lights.

4.1.2. Median filter

The second filter is the median filter based on the bubble sort of the $3*3$ neighboring pixels [24]. The median filter is a nonlinear digital filter which is able to preserve sharp signal changes and is very effective in removing impulse noise. This filter is widely used in digital signal and image/video processing applications. For the median filter, pixels are first sorted based on intensity. The center pixel would be the middle value of the sorted list of pixels. We present an example of ImpulseC code for the mean filter in Figure 1.

The filters are implemented by sliding a window of odd size (a $3*3$ window) over an $N*M$ image. At each window position, the sampled values are sorted and the resulting value of the sample replaces the sample in the center of the window. Three 32-bit streaming inputs provide four pixels for each line for each clock cycle. The size of the internal storage is $6*3$ pixels. Three internal storage solutions are implemented and evaluated. The first one is a sequential one with RAM as internal storage. The second one is a parallel/pipeline solution with RAM as

```
void FIL_MEAN(unsigned char image[][n],
unsigned char vimage[][vn])
{short x, y;
unsigned int sum;
   for (x = 1; x < vn − 1; x++)
      for (y = 1; y < vn − 1; y++)
      {
        sum = vimage[x][y];
        sum += vimage[x−1][y];
        sum += vimage[x + 1][y];
        sum += vimage[x − 1][y − 1];
        sum += vimage[x][y − 1];
        sum += vimage[x + 1][y − 1];
        sum += vimage[x − 1][y + 1];
        sum += vimage[x][y + 1];
        sum += vimage[x + 1][y + 1];
        image[x − 1][y − 1]= sum/9;
      }
```

(a) OriginalC code

```
while (co_stream_read(r0, & p0, sizeof(int32))
== co_err_none){
#pragma CO PIPELINE
#pragma CO set stageDelay 32
#pragma CO nonrecursive image0
#pragma CO nonrecursive image1
#pragma CO nonrecursive image2
co_stream_read (r1, & p1, sizeof(int32));
co_stream_read (r2, & p2, sizeof(int32));
for (k = 2; k < 6; k++){
#pragma CO UNROLL
image0[k]= p0 & 255; p0 = p0 ≫ 8; image1[k]= p1 & 255;
p1 = p1 ≫ 8;
image2[k]= p2 & 255;
p2 = p2 ≫ 8; }

res = 0;

for (k = 1; k < 5; k++){
#pragma CO UNROLL
res =((image0[k − 1]+ image0[k]+ image0[k + 1]
+ image1[k − 1]+ image1[k]+ image1[k + 1]
+ image2[k − 1]+ image2[k]+ image2[k + 1])≫ 3)
+(res ≪ 8);
}

for (k = 0; k < 2; k++){
#pragma CO UNROLL
temp = image0[k + 4];
image0[k]= temp;
temp = image1[k + 4];
image1[k]= temp;
temp = image2[k + 4];
image2[k]= temp; }
co_stream_write (output_stream, & res, sizeof(int32));
```

(b) ImpulseC code

Figure 1: The C and ImpulseC codes for a 3∗3 mean filter.

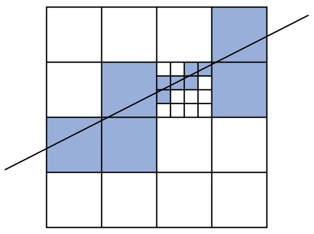


Figure 2: 2D recursive octree grid traversal principle.

internal storage. Three separate RAMs are used to allow parallelism between the three inputs. The third solution is a parallel/pipeline solution that uses registers as internal storage. Figure 2 represents the structure of external signals for both filters. Three streaming inputs are used to provide 4 pixels. Each input corresponds to one line of the image.

4.1.3. FFT

The third benchmark is the radix-4 FFT on 256 complex values (16-bit).

4.1.4. Ray casting in projective geometry (RCPG)

The ray casting in projective geometry (RCPG) is an iterative octree traversal algorithm (see Figure 2) [25]. In a regular grid, from a current cell crossed by the ray, the next cell is defined by a minimization of a cost function which is iteratively updated. At each step, the ray is propagated in 3D along the directions x, y, and z. The direction depends on the face where the ray and the current cell intersect (see Figure 2). The parameters of the intersection between the ray and each face are progressively stored and updated. The data structure is an octree structure: a cell can contain a data pointer to a higher resolution grid. Thus at each cell, the algorithm continues to the next one or descents in the subgrid. When it reaches a subgrid border, it mounts to the upper level of the grid.

For this design case, the sequential and pipeline algorithms are described.

The chosen cases are described in Table 3.

4.2. Target platform

The previously described case studies are intended to be synthesized, placed, and routed on a target technology in

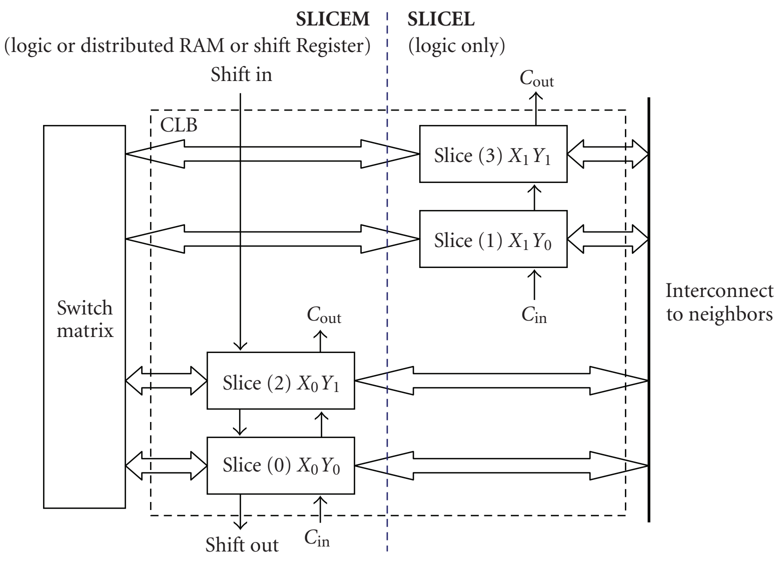


FIGURE 3: Virtex-4 slice structure. *Source: Xilinx Virtex-4 user guideUG070 (version 2.3).*

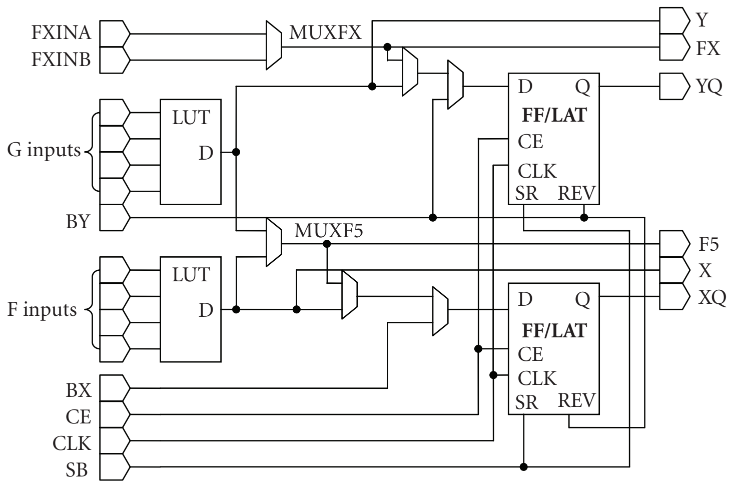


FIGURE 4: Virtex-4 slice L structure. *Source: Xilinx Virtex-4 user guideUG070 (version 2.3).*

TABLE 3: Core case studies.

Benchmark	Description	No IP
Linear digital filter (mean)	Pipeline using registers	IP1
	Pipeline using memory	IP2
	Sequential	IP3
Nonlinear digital filter (median)	Pipeline using registers	IP4
	Pipeline using memory	IP5
	Sequential	IP6
FFT	Sequential	IP7
	Pipeline	IP8
RCPG	Sequential	IP9
	Pipeline	IP10

order to evaluate how the selected C-based design environment outputs as inputs to a same synthesis, place, and route tool will be processed. The metrics to be considered will be performance and area.

4.2.1. Target technology

We selected the FPGA Xilinx Virtex-4 technology [26] as the target technology in this case study. The main reason for the choice of an FPGA technology was to allow a quick implementation and verification of all the IPs through actual execution on chip. The Virtex-4 technology based on a 90 nm copper CMOS process has a fixed number of hardcores resources such as DSP, embedded RAM, and fixed interconnections. The Xilinx Virtex-4 can be described as a matrix of CLB each of them being composed of several slices (see Figure 3). The Xilinx Virtex-4 proposes memory-oriented slices *SLICEM* and logic-oriented slices *SLICEL* (see Figure 4). Also the embedded memory BRAM is a dual port 18 kb memory array. Hard cores in the FX family include embedded processors—IBM PowerPC—and 18 × 18 two's complement signed multipliers (DSP blocs).

The FPGA structure represents an additional challenge for C- and SystemC- based synthesis tools due to the higher granularity and heterogeneity of FPGA compared to ASIC. The variety of FPGA resources makes the resource selection more difficult for the compiler tools to synthesize high-level C constructs. Several similar resources can be good candidates for one C-construct. The compiler tool has to select the most appropriate resource among all these candidate resources.

4.2.2. *Tools and options combinations*

Physical synthesis has been conducted using Xilinx tools ISE XST [27]. We conducted an intensive and wide automatic exploration of physical synthesis options (synthesis, place, and route). These range from area to speed oriented optimizations with in addition an exploration of the device density factor. All configurations have been executed on an FPGA board. This physical level design space exploration comes as a complement to high-level optimization techniques used by C-based synthesis such as for example, speculative execution (pipeline). The mutual effects—potentially inhibitory—of C-based synthesis followed by a VHDL physical synthesis are unspecified in any of the tool documentations. Different combinations of synthesis and place/route options on the different cases are explored in order to evaluate the possible interactions. The options used for VHDL synthesis and place and route are the global optimization options for area and speed with the level of optimization:

(1) XST VHDL synthesis option: -*opt_mode* speed or area,

(2) mapping option: -*cm* balanced or speed or area,

(3) place and route option: -*ol* std or med or high or -*ol* high -xe *n*.

This results in 24 combinations of synthesis, place and route options for each IP.

4.3. *Timing results*

Timing results are presented in Figure 5. The timing results are obtained for each IP with the use of the previously presented tools. The timing metrics are the clock frequency, the latency, and the number of cycles per result.

The variability of results between the tools comes from different reasons. Firstly, the RAM implementation is a direct implementation with no multiplexing of resources. The three RAMs of the filters are accessed with one access per clock cycle. It results in a limitation of the pipeline rate of twelve cycles per data produced. Secondly, the analysis of the results can be divided in two points. The first point concerns the different approaches of the used tools. For SystemC and Handel-C tools, pipeline needs to be explicitly described. The C-code is functional with no specific programing with the ImpulseC tool. The number of stages of the pipeline is not precisely controlled with ImpulseC but indirectly through timing constraints. The automatic exploration of different options and constraints is the only solution to obtain the best compromise between the different constraints as the impact of the throughput/latency of the

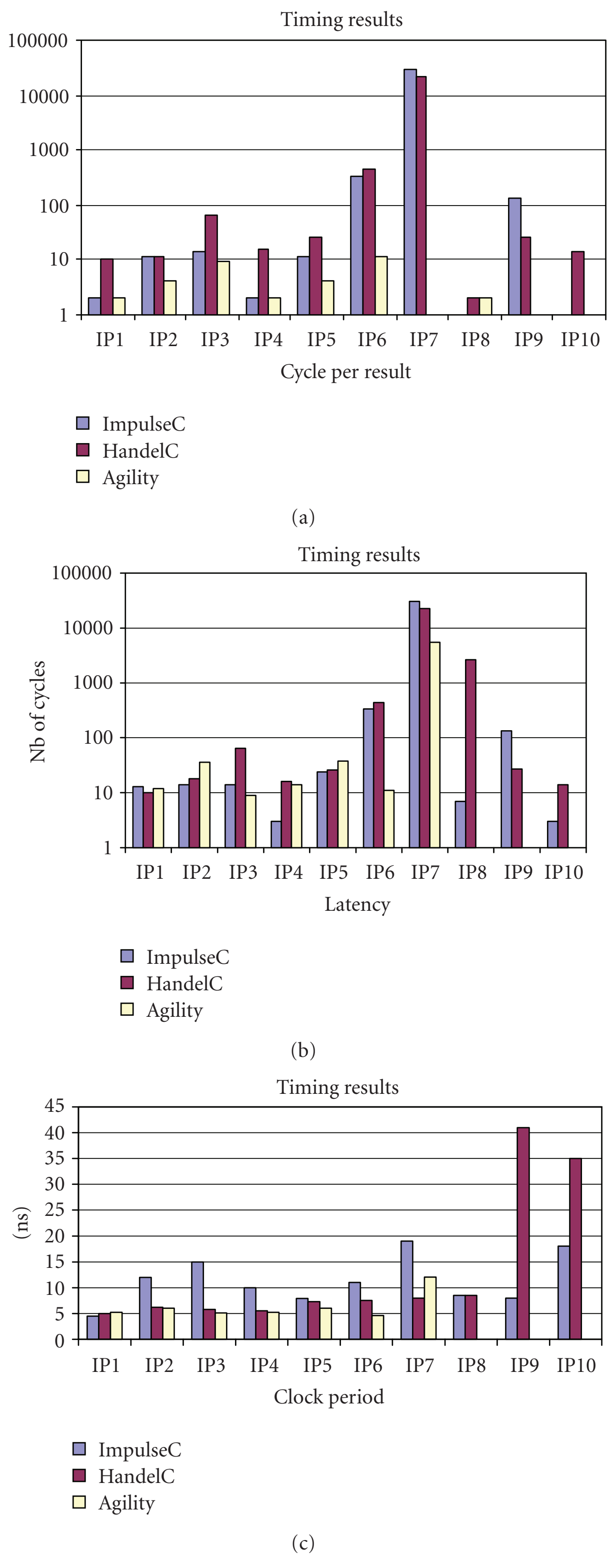


Figure 5: Timing results for throughput, latency, and clock period.

pipeline on the area/frequency is not straightforward. The difference of throughput between a pure sequential solution and a fully pipeline solution can be more than two orders of magnitude. This is the main source of performance/area

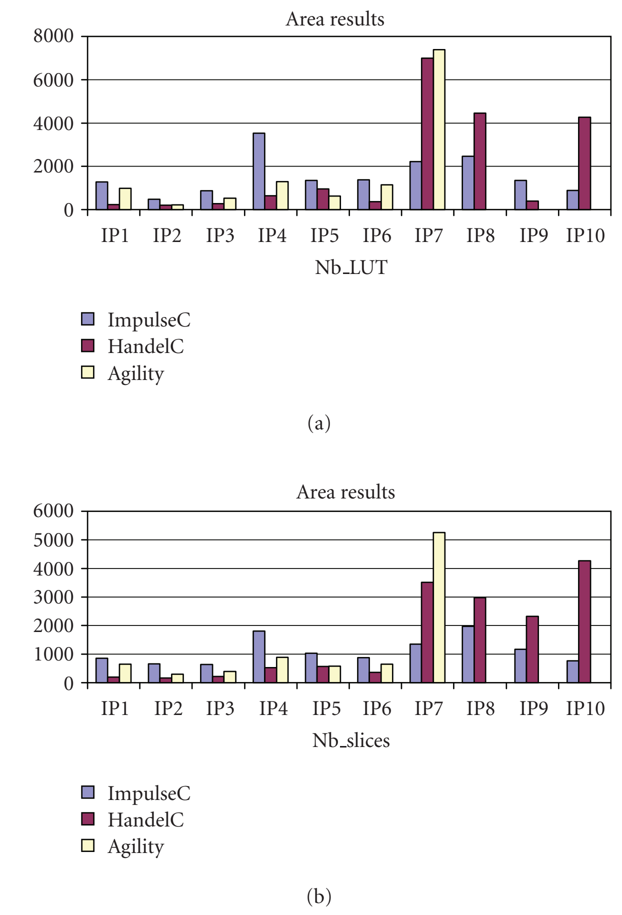


Figure 6: Area results for logic elements.

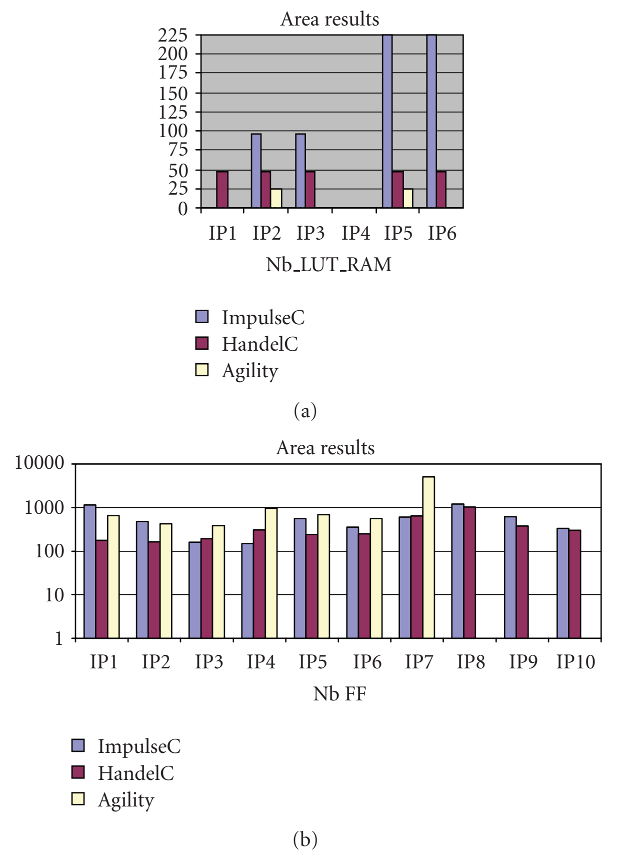


Figure 7: Area results for storage elements. (a) Number of LUT BRAM (the number is zero for IP 7 to IP 10) and (b) number of flip flop.

tradeoff at this level. This difference is increased with the implementation variability.

4.4. *Area results*

The area results have been obtained through VHDL generation of the various case studies followed by synthesis and place and route using Xilinx XST tool. They are presented in Figures 6 and 7. Our area metrics contain various resources present in the Xilinx Virtex-4 that are slices, flip flop, look-up table, and RAM. The synthesis and place and route options used are here the default options.

A first observation is that behavioral C-based approach with ImpulseC produces not always but often more logic and storage elements and a higher clock period than the other approaches. On the other side, ImpulseC brings the advantage of abstraction and genericity. This difference is not critical as throughput and latencies are similar.

4.5. *Variability of results with compilation options*

The results presented in Figures 8 to 12 show a variability of timing and area results according to the options used for synthesis and place and route. This variability depends on the front-end tool used (agility, DKDesign, and ImpulseC Codeveloper). The results presented in Figures 5 to 7 should be revisited for each option. These important clock period variations up to 150% are obtained with a variation of area cost between 100 and 200 slices, that is, 10 to 20% of area variation. The impact of tool options has a significant impact on timings compared to the impact on the size. Another major observation is the variability of results between options. This variability can be more significant than the variability between front-end tools. For instance the clock period variation is about 10 nanoseconds for the pipeline example (Figures 9 and 11). Thus ImpulseC gives better results with one option, for example, option 11 but not with another option. Agility gives better results with option 1 (Figure 9). In fact the best tradeoff is found by a careful analyze of the area/timing results. The area and timing results are not always linked as we can see with configurations 19 and 11 in Figure 8. For both configurations, the clock period is small but configuration 11 provides a higher area result compared to configuration 19. These variations also exist with placement constraints. The timing results can be either better or even worst when constraining area placement. Figure 12 compares the 24 configurations exploration with (left) and without (right) placement constraints. The improvement here is at most 0.15 nanosecond on the clock period which is not significant. Results were even worst

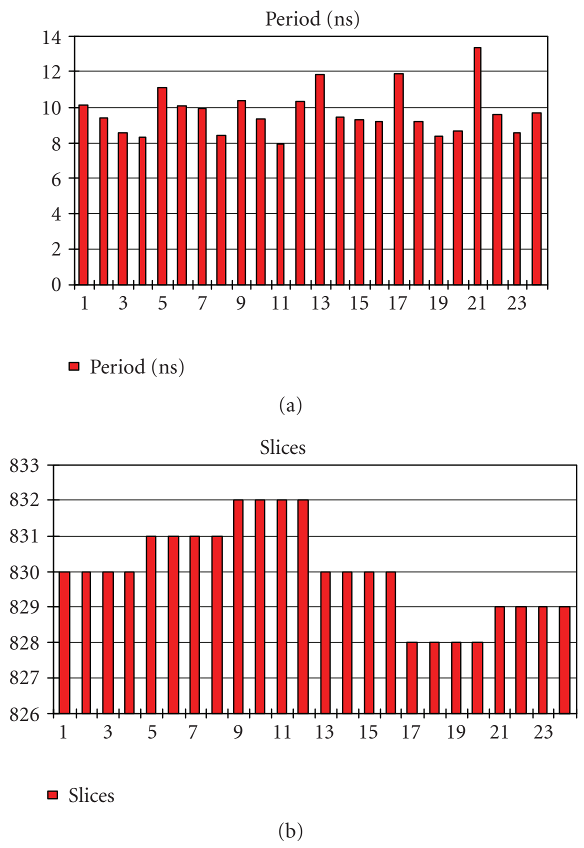


FIGURE 8: Sequential mean filter with impulseC. XST VHDL synthesis tool variation (a) period (b) slices.

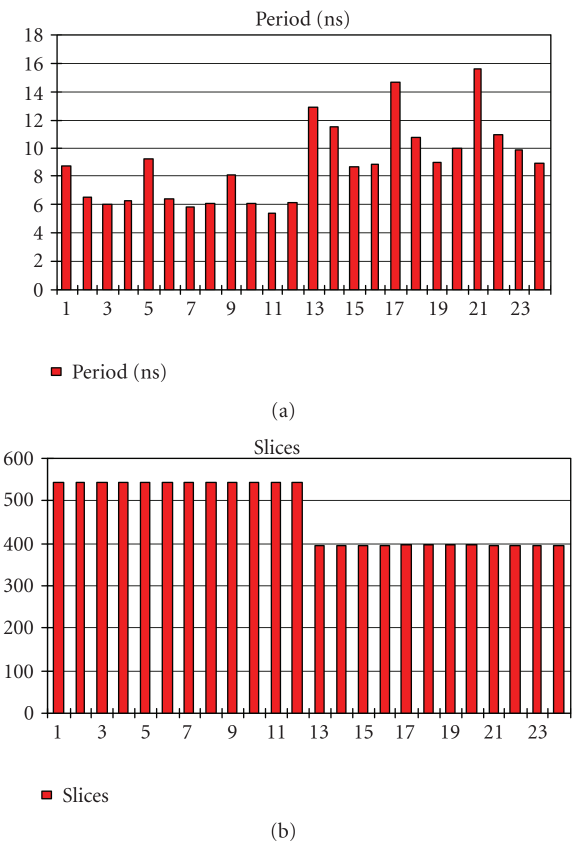


FIGURE 9: Pipeline mean filter with impulseC. XST VHDL synthesis tool variation (a) period (b) slices.

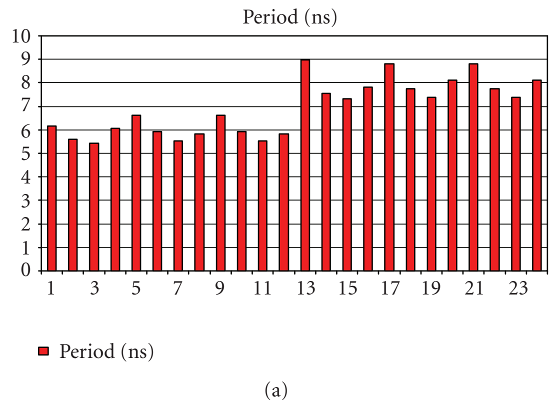
for the FFT. Thus a manual floorplanning becomes really difficult for heterogeneous hardware architectures such as FPGA. It should be reminded that obviously synthesis and place and route can incur large variations if no constraints are imposed and if large chips are selected. With large chips, the design can be spread without constraints conducting to higher delays. In our case, first the synthesis and place and route stages are done without constraints. Then a constrained place and route is used for the other configurations tested. The results are better with constraints.

The last point concerns the impact on place and route described on several examples in Figures 13 to 16 from best to worst. Best solutions are less spread and thus have reduced delays and higher operating frequencies.

The variations in terms of area and resources are obvious from the above figures. This points out that C-based synthesis may generate very different implementations resulting from C-based high-level modeling and the strong impact of backend tools. It should be noted that with heterogeneous devices such as Virtex-4 where hard cores are embedded the place and route tools may decide to implement a function in the vicinity of such embedded cores even if no interaction exists. This can be easily observed in Figures 13 and 14 (right part). This affects the quality of the implementation as the logic is spread out on the circuit. This clearly shows that there is a missing link between the system modeling level and the physical implementation. A feedback is necessary to help joint optimization.

4.6. Discussion

The C-to-hardware compilers considered here take two approaches to concurrency. The first approach chosen by Handel-C and agility compiler adds parallel constructs to the language. It forces the programmer to expose most concurrency that is not a difficult task in major cases. Handel-C provides specific constructs that dispatch collections of instructions in parallel. These additional statement constructs can be used by any programmer. The compilers considered here use a variety of techniques for inserting clock cycle boundaries. Handel-C and agility use fixed implicit rules for inserting clocks and are very simple. Assignments and delay statements each takes one cycle in Handel-C and instructions between two wait() statements take one cycle in agility SystemC. All the instructions inserted in a *par* statement are executed in one clock cycle in Handel-C. For all the implemented filters, adding manually parallelism is an easy task that can be achieved by any programmer. On the other hand, pipeline extraction can become a tricky task as algorithm must be written in that way. An example was the FFT algorithm implementation: adding pipeline from a sequential code can take a long time and changes



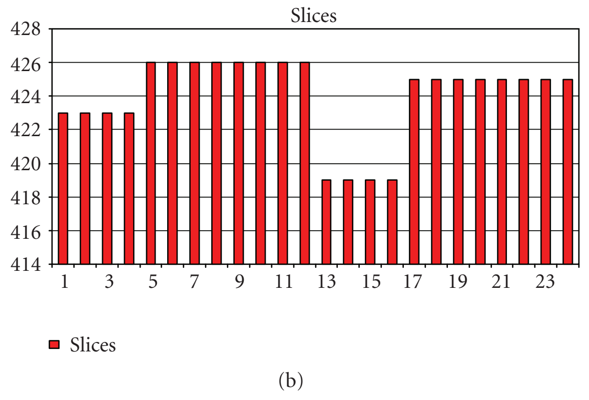
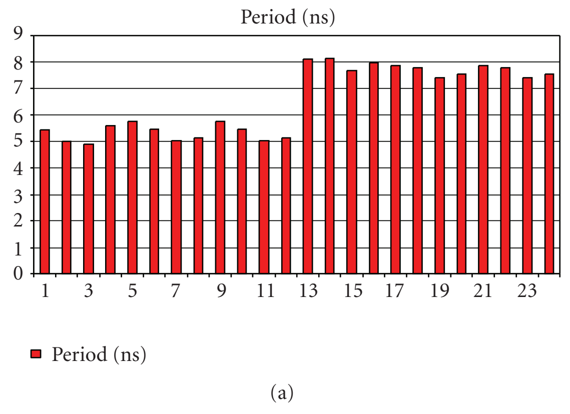


FIGURE 10: Sequential mean filter with agility. XST VHDL synthesis tool variation (a) period (b) slices.



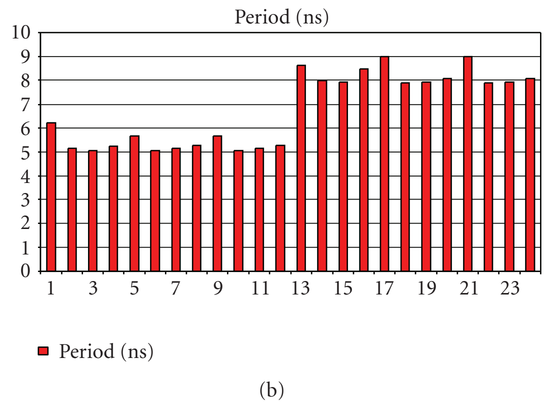
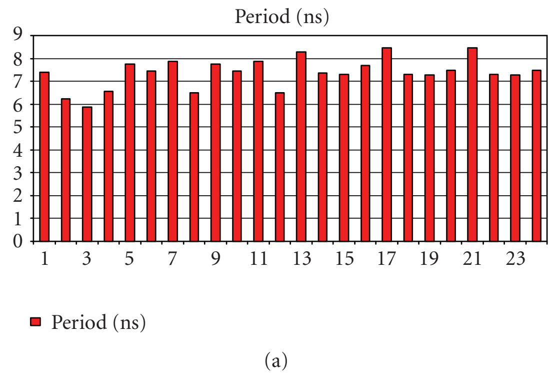


FIGURE 12: Pipeline median filter with agility. XST VHDL timing variation with and without placement constraints.



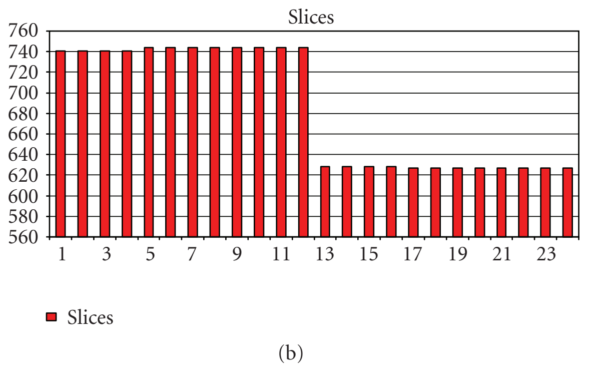


FIGURE 11: Pipeline mean filter with agility. XST VHDL synthesis tool variation (a) period (b) slices.

are important to make. It is even more difficult to express pipeline with Handel-C than SystemC as dependencies between instructions impose the use of different cycles. The precise control of logic/operators per clock cycle is difficult with Handel-C: either all the instructions in one stage are independent and the pipeline clock can be of one clock cycle per result or reuse is possible that makes the number of clocks per result proportional to the reuse rate. Another solution is to use a higher frequency and divide the processing in elementary cycles (one per code line). SystemC agility compiler representation becomes therefore almost an RTL level representation allowing optimization at the clock cycle level.

The second approach lets the compiler identifies parallelism helped with pragmas in the source code. ImpulseC compiler allows automatic pipelining through pragmas but only for inner loops. Loop unrolling is used to obtain full pipelining. Precise control of the number of stages is difficult with such pragmas. These simple rules can make it difficult to achieve a particular timing constraint. It is difficult to predict in advance when a second input data can be inserted, that is the throughput. Several synthesis cycles must be operated to converge to the best solution. The tool helps this exploration by automating the use of VHDL synthesis tool in the loop. Pipeline exploration is conducted automatically with VHDL synthesis on different solutions providing a frequency graph function of the latency/rate of the pipeline. This helps to obtain the higher rate/latency pipeline but with

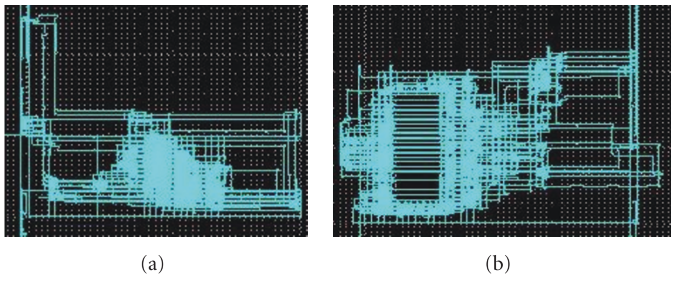


Figure 13: Sequential mean filter. Place and route variations from best (a) to worst (b).

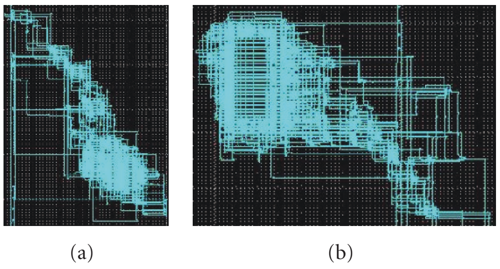


Figure 14: Pipeline mean filter. Place and route variations from best (a) to worst (b).

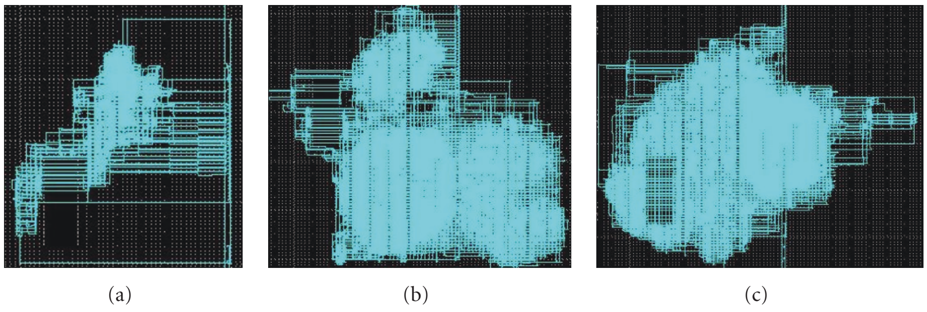


Figure 15: Pipeline FFT. Place and route variations from best (left) to worst (right).

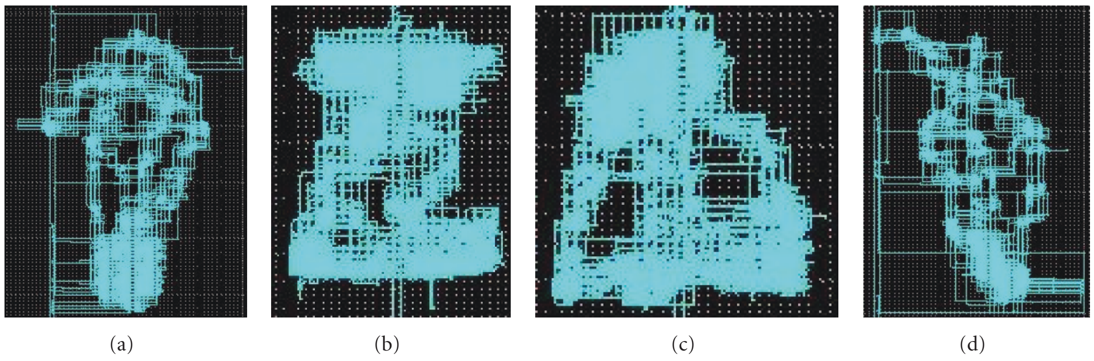


Figure 16: Pipeline median filter. Place and route variations from best (left) to worst (right).

no considerations of the area. It is thus difficult to make a compromise between timing and area constraints.

The IP interfaces provided by ImpulseC are FIFO or memory which is better adapted to stream processing. In fact, it is difficult to design specific protocols at RTL level. Also considering local memory storage, RAM/register inference selection is only obtained through a compilation option of ImpulseC, that is for all the design and not separately for each array, which is really limiting as registers are a limited resource in FPGA. The two other tools provide pragmas to define precisely which way to store arrays of data, with registers or RAM.

According to the data types, C-based design tools considered several approaches. The first approach neither modifies nor augments C's types but allows the compiler to adjust the width of the integer types outside the language. The second approach is to add hardware types to the C-language. Handel-C and ImpulseC compilers chose the data customization. The programmer cannot cast a variable or expression to a type with a different width, that makes the code more difficult to write. For the filter implementation, arrays are of several sizes and the indexes' sizes are different. The programmer must often use the concatenation operator to zero pad or sign extend a variable to a given width that makes the programing time longer.

Most of the Handel-C debugging time was spent in adjusting the size of data and manually programing the pipeline optimization. The programmers must carefully analyze the code to specify all the widths and it can quickly be tiresome. A parser for automatic adjusting of the size of any used variable according to the type of the operators used can be considered.

One main argument to choose an approach to use is the level of description needed at the interface level of the design. If FIFO or memory-like protocols are sufficient, behavioral C-based HLS is now a mature solution with equivalent performances and area results than a more precise almost RTL level C-based solution like agility compiler or Handel-C. Furthermore, behavioral C-based HLS provides abstraction and genericity of the pipeline allowing easy retargeting of hardware in different timing/area constraints without redesigning. This criterion is fundamental in streaming applications where throughput is the key performance parameter.

5. CONCLUSIONS AND FUTURE WORK

We have conducted a case study on the evaluation of C-based high-level synthesis systems. The objective was to assess a potential higher use of this in-area constrained high-level system multiobjective partitioning and how system decisions could be impacted. Indeed, although growing system complexity calls for high-level abstract modeling, it is still mandatory to take into account precise implementation feedback to improve performances. This puts into question the capacity of C-based tools to meet this challenge. Evidences on case studies of significant result variations among the high-level synthesis tools and their emphasis through physical synthesis options exploration challenge the use of C-based multiobjective modeling methodologies for system design. In a multiobjective approach, area-performance system design tradeoffs should be based on as accurate as possible data otherwise inappropriate design decisions could be made. We argue that implementation issues (area, frequency, and floorplan) for large scale complex systems should be taken into account when using C-based high-level modeling since currently the tools do not guarantee that high-level concurrency semantics properties are preserved. Indeed, extracted concurrency at high level is challenged through code and representation transformation as well as resources constraints.

Solution to this comes through an integrated flow with concurrency properties propagated as constraints as well as concurrency feedback to the highest level.

Future work will extend the size of the case studies and automate the evaluation process.

ACKNOWLEDGMENT

The authors are grateful to the reviewers for their excellent reviews which helped improve the paper dramatically.

REFERENCES

[1] ITRS 2007 Edition—System, http://www.itrs.net/.

[2] IEEE 1666 Standard SystemC Language Reference Manual, http://www.systemc.org/.

[3] C. Haubelt, J. Falk, J. Keinert, et al., "A systemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 22 pages, 2007.

[4] S. Ouadjaout and D. Houzet, "Generation of embedded hardware/software from systemC," *EURASIP Journal on Embedded Systems*, vol. 2006, Article ID 18526, 11 pages, 2006.

[5] M. O. Cheema, L. Lacassagne, and O. Hammami, "System-platforms-based systemC TLM design of image processing chains for embedded applications," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 71043, 14 pages, 2007.

[6] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '95)*, pp. 136–144, Napa Valley, Calif, USA, April 1995.

[7] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, "Cyber"," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '99)*, pp. 390–393, Munich, Germany, March 1999.

[8] D. C. Ku and G. De Micheli, "Hardware C: a language for hardware design," Tech. Rep. CSTL-TR 90-419, Computer System Laboratory, Stanford University, Stanford, Calif, USA, August 2000.

[9] T. Kambe, A. Yamada, K. Nishida, et al., "A C-based synthesis system, Bach and its application," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 151–155, Yokohama, Japan, January-February 2001.

[10] E. Grimpe and F. Oppenheimer, "Extending the systemC synthesis subset by object-oriented features," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 25–30, Newport Beach, Calif, USA, October 2003.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the 16th International Conference on VLSI Design*, pp. 461–466, New Delhi, India, January 2003.

[12] V. S. Saun and P. R. Panda, "Extracting exact finite state machines from behavioral systemC descriptions," in *Proceedings of the 18th International Conference on VLSI Design*, pp. 280–285, Kolkata, India, January 2005.

[13] H. D. Patel, S. K. Shukla, and R. A. Bergamaschi, "Heterogeneous behavioral hierarchy extensions for systemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 765–780, 2007.

[14] S. Chtourou and O. Hammami, "SystemC space exploration of behavioral synthesis options on area, performance and power consumption," in *Proceedings of the 17th International Conference on Microelectronics (ICM '05)*, pp. 67–71, Islamabad, Pakistan, December 2005.

[15] ROCCC, http://www.cs.ucr.edu/~roccc/.

[16] Synopsys, "Behavioral Compiler User Guide Version 1999.10," 1999.

[17] Agility,http://www.agilityds.com/products/default.aspx.

[18] Forte Design, http://www.forteds.com/.

[19] Orinoco Dale, http://www.chipvision.com/company/index.php.

[20] ImpulseC Inc, "Co-developper's user guide," 2007, http://www.impulsec.com/.

[21] CatapultC, http://www.mentor.com/.

[22] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.

[23] Handel-C, http://www.celoxica.com/.

[24] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[25] J. Revelles, C. Ureña, and M. Lastra, "An efficient parametric algorithm for octree traversal," in *Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '00)*, Plzen-Bory, Czech Republic, February 2000.

[26] Xilinx Virtex-4, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.

[27] Xilinx ISE 9.2, http://www.xilinx.com/ise/logic_design_prod/foundation.htm.