Des Autom Embed Syst
DOI 10.1007/s10617-012-9096-8

# An overview of today's high-level synthesis tools

**Wim Meeus · Kristof Van Beeck · Toon Goedemé · Jan Meel · Dirk Stroobandt**

Received: 23 November 2011 / Accepted: 2 August 2012


**Abstract** High-level synthesis (HLS) is an increasingly popular approach in electronic design automation (EDA) that raises the abstraction level for designing digital circuits. With the increasing complexity of embedded systems, these tools are particularly relevant in embedded systems design. In this paper, we present our evaluation of a broad selection of recent HLS tools in terms of capabilities, usability and quality of results. Even though HLS tools are still lacking some maturity, they are constantly improving and the industry is now starting to adopt them into their design flows.

**Keywords** Electronic system-level · High-level synthesis

## 1 Introduction

High-level synthesis (HLS) is a new step in the design flow of a digital electronic circuit, moving the design effort to higher abstraction levels. Its place in the design flow can be best situated using Gajski and Kuhn's Y-chart [9]. This chart (Fig. 1a) has 3 axes that represent different views on the design: behavior (what the (sub)circuit does), structure (how

W. Meeus (✉) · D. Stroobandt
Electronics and Information Systems, Ghent University and imec, Ghent, Belgium
e-mail: Wim.Meeus@UGent.be

D. Stroobandt
e-mail: Dirk.Stroobandt@UGent.be

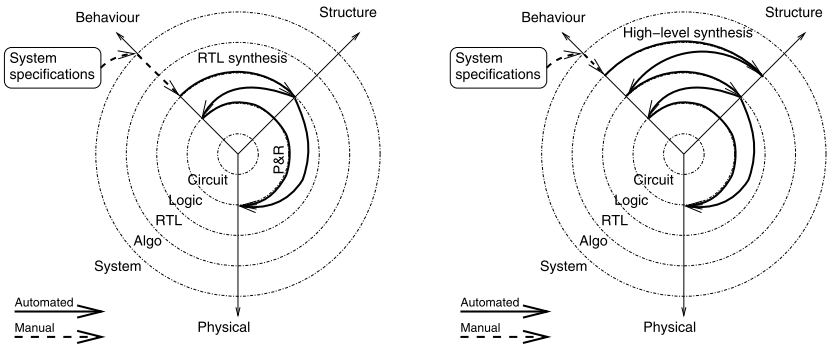
K. Van Beeck · T. Goedemé · J. Meel
Campus de Nayer, Lessius Mechelen, Sint-Katelijke-Waver, Belgium

K. Van Beeck
e-mail: kristof.vanbeeck@lessius.eu

T. Goedemé
e-mail: toon.goedeme@lessius.eu

J. Meel
e-mail: jan.meel@lessius.eu

Published online: 21 August 2012




(a) RTL design flow (b) HLS design flow

**Fig. 1** High-level synthesis in the Gasjki-Kuhn Y-chart

the circuit is built together, e.g. a netlist or schematic) and geometry (how the circuit is physically implemented or what it looks like, e.g. a layout). There are also 5 concentric circles representing abstraction levels: circuit (transistors, voltages, circuit equations), logical (ones and zeros, boolean equations), register transfer (RTL: registers, operators, HDL), algorithmic (functions, loops, programming languages) and system level (system specifications, programming and natural languages).

In the design flow that has become mainstream in the past 2 decades (Fig. 1a), the hardware designer would manually refine the *behavioral* system specifications down to the RT level. From that point, *RTL synthesis* and *place and route* complete the design flow. As both the RTL design and design tools are made by humans, *verification* is necessary to match the behavior of the design with the specifications at various stages of the design flow and remove discrepancies when necessary.

Nowadays this design flow is increasingly being challenged. Moore's law [14] states that an increasing amount of functionality can be integrated on a single chip. Someone has to design all this functionality though, and it is not economically nor practically viable to steadily increase the size of design teams or design time. This means that somehow the *design productivity* has to be improved. Given the fast growing transistor count, it may be acceptable to trade some transistors (or chip area) for an improved design time.

High-level synthesis improves design productivity by automating the refinement from the algorithmic level to RTL. As can be seen on the Y-chart (Fig. 1b), the transition from the specifications to the start of the automated design flow becomes smaller now. HLS generates an RTL design from a function written in a programming language like C, C++ or Matlab.

High-level synthesis takes over a number of tasks from the designer. After analysis of the source code, *resource allocation* is done, i.e. determining what types of operators and memory elements are needed and how many. Next, during *scheduling*, each operation from the source code is assigned to a certain time slot (clock cycle or *cstep*). Finally, in *resource binding*, operations and data elements from the source code are assigned to specific operators and memory elements. High-level synthesis also takes care of *interface synthesis*, the generation of an appropriate interface, consisting of data and control signals, between the generated circuit and its periphery (e.g. a memory interface).

Several advantages arise from the use of HLS in the design flow. First of all, the amount of code to be written by designers is reduced dramatically, which saves time and reduces the risk of mistakes. HLS can optimize a design by tweaking source code and tool options,

opening up opportunities for extensive design space exploration. Verification time, which nowadays exceeds design time, is reduced a lot because the HLS tool can, in addition to the design itself, generate testbenches, thereby reusing test data that was used to validate the source code.

This is particularly relevant for the design of FPGA based embedded systems. Moving to a higher abstraction level makes it possible to handle the increasing design complexity while removing the need to hand-code the hardware architecture and timing into the algorithm. Hardware accelerators for embedded software may be generated with minimal effort. HLS and FPGAs make a perfect combination for rapid prototyping and a fast time to market.

In this paper we give an overview of a wide selection of HLS tools that are currently available. One has to bear in mind that this evaluation is a current snapshot of tools available today. Tools are constantly changing as HLS is a very active research area. We hope that this paper provides an initial status and that tool developers will be motivated by it to work on current shortcomings.

The paper is organized as follows. In Sect. 2, we discuss a number of recent papers on the topic. Section 3 presents our evaluation criteria, and in Sect. 4 a simple test application is introduced. The main part is Sect. 5 in which the different HLS tools are discussed. Section 6 describes the design of a larger application and in Sect. 7 we draw a number of conclusions.

## 2 Related work

As explained above, HLS is gaining importance in the design of digital electronic circuits. [4] summarizes the main benefits of HLS: it allows the designer to handle the increasing design complexity, reduces verification effort, allows for design space exploration and design optimization for various criteria (such as power or latency) and helps in handling process variability.

[13] discusses the history of HLS and identifies a number of reasons why the current generation of HLS tools may be successful where earlier attempts failed: a.o. input languages that are used in algorithm design, improved quality of results and the rise of FPGAs. At DAC 2010, a panel discussed the input language question [6]. From the summary report it appears that the discussion was limited to tool vendors advocating their language of choice.

A number of papers, such as [17], present designer's experiences with HLS tools. Their work showed that since HLS is still evolving, it is necessary to experiment with multiple tools for the same problem. The quality of the overall result does not only depend on the HLS tool used but also on the integration with the downstream design flow and design libraries. [10] evaluates academic HLS tools. These tools are interesting from a research point of view, but a lack of manpower in academia to support and consolidate them (particularly in the longer term) often makes them risky to use in a commercial environment.

Reviews of commercial HLS tools have been published in a number of papers. The company BDT, Inc. (www.bdti.com) certifies HLS tools and publishes reports on their website, such as [2] and [1]. [5] introduces the concept of HLS and explains how this flow is implemented in Catapult C (C++ based) and Cynthesizer (SystemC based). In [3], AutoPilot is used to highlight how recent evolution in HLS tools makes them viable for industrial adoption, particularly for FPGA design. Three more commercial tools are reviewed from a quantitative point of view in [12]. The number of tools in each paper is limited to a small number (maximum three), so a broad overview of the HLS market is still missing, as well as the user perspective: learning curve, amount of effort to write source code, etc. In this paper, we provide a comprehensive evaluation of many HLS tools, including quality of experience (QoE) aspects.

## 3 Evaluation criteria

In our work, we have evaluated the selection of HLS tools for a broad set of criteria, encompassing design entry (source language, documentation, code size), tool capabilities (support for data types, optimization), verification capabilities and quality of results. A spider web diagram as in Fig. 2 summarizes most of these criteria. We will use them further on to compare the HLS tools in a graphical way.

*Source language and ease of implementation* One of the key assets of HLS is that it could bridge the gap between algorithm design and hardware design, thus opening up hardware design to system designers that have until now no hardware design experience. It is preferred that hardware design can start from the code written by the algorithm designer rather than having to re-implement the design in a different language. It is not desirable to force algorithm designers into using a hardware oriented language that may lack expressiveness and flexibility [7].

It is obvious that certain restrictions on a high level programming language may be necessary for the code to be implementable as hardware. For instance, in order to generate the correct amount of memory and related address bits, a tool may require that arrays must be passed and accessed as arrays and not as pointers, and that the size of any array must be known at compile time. However, restrictions should be as loose as possible, not only because (as B. Stroustrup said) *the connection between the language in which we think/program and the problems and solutions we can imagine is very close*, but also because an overly restrictive design language may make programming a certain behavior very hard (and frustrate the designer during the process).

*Tool complexity, user interface and documentation* For wide adoption, a HLS tool should have a reasonably flat learning curve. This can be achieved a.o. with an intuitive graphical user interface that guides the designer through the flow. More complex tool features, as well as a scripting interface for batch processing, should be available for the advanced user but, at the same time, default tool settings should give less experienced designers a head start. Clear documentation is essential too, not only on how to use the tool but also on how to write and fine-tune source code.

*Support for data types* In software, available data types are usually the supported types of the instruction processor (int, float etc.) and aggregates thereof. In hardware, however, the only primitive data type is a single bit. Direct support for more complex data types by RTL synthesis tools is usually limited to integers. The hardware designer has full freedom to determine ranges of these integers (no. of bits). A good HLS tool should allow the designer to make this choice and let them validate this choice at the source code level. Support for additional data types such as fixed point or floating point is also an asset as it eases the transition from algorithm to RTL design.

*Design exploration capabilities* A major strength of HLS is its design exploration capabilities. Rather than making architecture decisions upfront and coding RTL by hand, HLS allows evaluation of a number of architectures to choose the best one with respect to the design specifications. HLS tools differ largely in the way they guide this exploration process. For clarity and portability, maximal separation between the source code (behavior) and design constraints (architecture) is preferred.

*Verification* Given the size and the complexity of today's designs, verification is a major task in the design cycle. HLS has several opportunities to ease the verification engineer's task. Clear (sub)design interfaces that are consistent throughout the design flow are helpful to interface the generated design with a testbench.

A HLS tool can also speed up verification by generating testbenches together with the design. Test vectors used to validate the source code can be reused to validate the design. The most advanced automation of functional verification can be achieved by integrating the source code (as a golden reference) and the generated design (as a device under test) into one testbench. In that case, the simulator can apply the same stimuli to both reference and DUT and report discrepancies.

*Correctness* Some HLS vendors claim that their tool generates designs that are *correct by construction*. Correctness of a generated design obviously depends on the correctness of the tool. A more prudent view would be that a tool generates a design in a systematic way. If the tool is correct, the design will be correct as well, and a generated design that passes verification also validates the correctness of the tool. In this way, the probability of errors in a generated design is lower than in handcrafted RTL code.

*Generated design: size, latency, resource usage after synthesis* High-Level Synthesis usually leaves the designer with a RTL design. The latency of this design is a fixed number of clock cycles, as determined by the scheduling step of HLS. The achievable clock rate, and hence the overall design latency, is still unknown at this point, as well as the final size of the design in terms of sq.mm (ASIC) or resource usage (FPGA). Taking a HLS design through RTL synthesis gives good (if not exact) figures of these properties, which in turn gives an idea of the quality of the generated design.

During design exploration with HLS it is possible to generate different RTL designs by tuning HLS constraints and source code. After RTL synthesis, each one can be represented as a point in a (area,latency) graph and a Pareto optimal front can be constructed.

In our conclusion, a summary of each tool will be presented graphically as a spider web diagram. In these diagrams, similar to Fig. 2, the top-left axes represent the user experience, the top-right represents design implementation capabilities, the right hand side and bottom axes highlight the tool capabilities, and the bottom-left axis shows the quality of the result in terms of area. The quality of a tool is better for a certain metric if the spider web is closer to the outside of the corresponding radial axis.

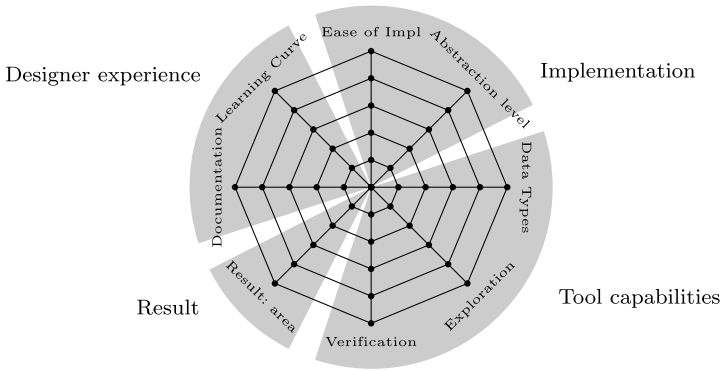



**Fig. 2** Spider web diagram

```
for (r : 1..rows)
  for (c : 1..cols)
    if (pixel_in[c,r] on image border)
      pixel_out[c,r] = white
    else
      gradX = pixel_in[c-1,r-1] + 2*pixel_in[c-1,r] + pixel_in[c-1,r+1] \\
            - pixel_in[c+1,r-1] - 2*pixel_in[c+1,r] - pixel_in[c+1,r+1]
      gradY = pixel_in[c-1,r-1] + 2*pixel_in[c,r-1] + pixel_in[c+1,r-1] \\
            - pixel_in[c-1,r+1] - 2*pixel_in[c,r+1] - pixel_in[c+1,r+1]
      grad = abs(gradX) + abs(gradY)
      if (grad>255) grad = 255
      pixel_out[c,r] = 255 - grad
    end if
```

**Fig. 3** Sobel Edge Detection: pseudocode

## 4 Test application: Sobel edge detection

For the evaluation of HLS tools, we had to choose a test application that was not trivial but at the same time simple enough to be implemented with reasonable effort. A good balance was found in the Sobel edge detector [11]. This image processing algorithm (pseudocode in Fig. 3) detects edges in a bitmap image by calculating the horizontal and vertical color gradient in a $3 \times 3$ window around each pixel. The sum of the absolute values of these gradients determines the shade of gray in the output pixel. A higher value of the sum means a sharper transition in the input value and a darker pixel in the output image. Border pixels in the generated image remain white.

This algorithm, even though relatively simple, already leaves a lot of opportunities to explore the capabilities of HLS tools. The sliding window is implemented with two nested loops, and there are no data dependencies between any two loop iterations. This means that iterations can be executed sequentially or with any number in parallel. Most input pixels are used in multiple (typically 8) loop iterations, an opportunity for internal reuse of data within the circuit and a substantial reduction of the input data bandwidth.

## 5 Tools overview

This section provides an overview and a qualitative evaluation of the different tools. A comparison is made in the concluding section (Table 1) and a graphical view of this evaluation is presented for each tool in a spider web diagram (Figs. 4 and 5). The tools are listed in alphabetical order. We are aware that more tools exist, but we had to make a selection based on relevance while taking the availability of software licenses into account.

### 5.1 Xilinx AccelDSP

AccelDSP is a HLS tool from Xilinx. It is based on the AccelChip DSP technology, which was acquired by Xilinx in 2006.

AccelDSP starts from an untimed Matlab description, and uses its own GUI. It only works on streaming functions, which the designer needs to code explicitly. This reduces the application domain to mainly image and signal processing applications. Only a limited part of standard M-code is supported in this tool. Automatic HDL code generation is performed by scanning the Matlab code and instantiating basic building blocks. This is similar to how

Xilinx System Generator synthesizes HDL code. It has an intuitive GUI, but no extensive tutorials exist to our knowledge.

The designer starts by writing a floating-point *golden reference* Matlab model. Using M-code as input, we found it very convenient to create and manipulate data arrays, such as line buffers for the Sobel edge detector. By dynamically estimating the needed bit width, automatic conversion from floating point to fixed point is performed. The fixed-point bit width is limited to 53 bits. When more bits are needed one needs to address the fixed-point toolbox. During this fixed-point generation, several statistics about the fixed-point implementation are calculated. After fixed-point conversion, the designs are compared. When necessary, the designer may add directives. Our experiments showed that manual tweaks are still necessary to prevent overflow.

AccelDSP allows for several design space exploration possibilities such as (partial) loop and matrix multiplication unrolling, pipelining and memory mapping, which is a major difference when compared to Xilinx System Generator. We tested the design space exploration capabilities of AccelDSP with our Sobel filter implementation. Our initial implementation ran at 61 MHz. When enabling the pipelining option, the design took about 4 % more resources, but achieved a clockspeed of 158 MHz.

Besides the automatic fixed-point conversion AccelDSP allows the use of *fixed-point probes*. These are specific Matlab commands which are inserted manually, so that the designer can plot the value of internal signals in time. For bit-level verification, AccelDSP automatically generates a HDL testbench. After compilation, the design can be mapped onto an FPGA for hardware-in-the-loop simulation.

After all fine-tuning is finished, three types of generations are available: either RTL code is generated, a Xilinx System Generator block is created, or the design can be simulated using hardware-in-the-loop. The latter is achieved by invoking System Generator. The generated RTL module communicates with the surrounding hardware using a handshake interface. The module requests and produces data using synchronous control signals.

We found that AccelDSP is a very powerful tool, but only for streaming data applications. The Matlab code needs to be rewritten manually into a streaming loop. The automatic fixed-point conversion is an interesting feature, but manual directives are still needed. There are many design space exploration possibilities, and besides RTL generation one can also generate a Xilinx System Generator block.

### 5.2 Agility compiler

Agility Compiler is a tool by Agility Design Solutions (ADS). Since January 2008 the software department of Celoxica joined with Catalytic to form ADS. In January 2009 ADS was acquired by Mentor Graphics. They concentrate on the production of Handel-C and SystemC tools.

Agility Compiler is a tool which is used to synthesize SystemC. It offers complete support for the OSCI SystemC synthesizable subset, and some additional Agility hardware features. Automatic code generation is supported for Actel, Altera and Xilinx FPGAs. Agility Compiler uses SystemC in a way that the designer still needs to manually write hardware processes, define sensitivity lists, etc. The main advantage over standard HDL entry is that hardware is described using *transaction level models*. The idea behind this approach is to separate the details of the communication between the modules from the module implementations, and to model them as channels. These channels are communication interfaces, for example FIFOs or data busses. Transactions are handled by calling the interface functions of the modules. All low-level data exchange information is hidden behind these functions.

TLM thus puts more emphasis on the data itself, and less on the actual implementation. Using these channels data exchange is very intuitive and straight-forward.

For verification purposes, we still need to manually write a SystemC testbench. No automatic testbench generation is available.

Synthesis optimizations such as fine grained logic sharing, re-timing, re-writing and automatic tree balancing are available. SystemC is a C++ class library, with an event-driven simulation kernel. The additional Agility hardware features include black boxes in which existing HDL IP can be imported, global asynchronous reset signals and signals with default values. The latter are signals that, if changed, return to their default value after one clock cycle. After each step extensive XML reports concerning area estimation and synthesis optimizations are generated. Agility Compiler also supports the automatic generation of control and data flow graphs. We found however that these seem to be complex for relatively simple hardware designs.

Based on our experience we can conclude that the main advantage of Agility Compiler is found in the use of channels: these allow for an efficient communication. One can also benefit from specific C++ functionality features such as operator overloading. Besides the use of these channels, we found that there is no other implicit abstraction available, thus one still stays at a rather low-level hardware description.

### 5.3 AutoPilot

AutoPilot was developed by AutoESL, which has recently been acquired by Xilinx. It compiles HDL from a variety of C-based input languages: ANSI C, C++ and SystemC. Only minimal C-code modifications are necessary to generate an RTL design. Autopilot can optimize for area, latency and power consumption. Recently, Xilinx revealed *Vivado ESL* as the new name for this tool.

Starting from generic C code, each datatype has to be converted into AutoPilot's arbitrary-precision datatypes. From there, algorithm and interface synthesis are performed on the design. AutoPilot's first synthesis step creates and explores the architecture, the second step builds the actual implementation and implements interfacing details. AutoPilot extracts the control and datapath behavior: the control is located in loops and conditions which are mapped to an FSM. Datapath evaluation is achieved by loop unrolling and evaluation of conditions. Based on user constraints e.g. for latency and resource usage, scheduling and binding are determined.

We experienced that indeed only minimal modifications are needed to generate RTL. Our Sobel filter C reference design consisted of 43 lines of code. A total of about 2100 lines of VHDL code were generated. The generated code was comprehensible but complex variable names made it difficult to read. AutoPilot supports the generation of several interface types, a.o. with or without handshake. To enable concurrency, AutoPilot provides automatic pipelining for functions and loops. Design exploration is possible using (partial) loop unrolling. After generation of the code, a design report is generated which gives information about the estimated clock period, the latency (best/average and worst-case) and a resource estimation.

For simulation and verification, AutoPilot allows the reuse of the C-code testbench by automatically generating a wrapper that enables communication between the generated RTL and the C testbench. Reusing the C testbench for RTL simulation significantly reduces the verification time. Floating to fixed-point conversions need to be done manually.

### 5.4 BlueSpec

BlueSpec, a tool by BlueSpec Inc., uses a different approach towards abstraction. The design is entered in BlueSpec System Verilog (BSV), a Verilog based language in which design modules are implemented as a set of rules. This forces the designer to fully re-implement algorithms to obtain a synthesizable design, no pre-existing code can be reused.

The rules are called Guarded Atomic Actions. They define independent actions and are coded as behavioral expressions. Since they are independent of each other, the BSV code stays very legible. Clock and reset signals are implicit, i.e. they are not visible to the designer. Scheduling and dependencies are handled implicitly by the BlueSpec compiler. Rules are scheduled based on the actions that they perform, together with their control signals. Using pragmas explicit rule scheduling can be performed.

All design elements are implemented as separate modules, which can lead to long portmaps and connection lists. Data exchange between modules is implemented in an object oriented fashion using interface methods, which is a very straightforward approach. Their interface methods implicitly generate multiple RTL ports. For specific hardware components, such as BlockRAM, Bluespec uses a server/client interfacing approach. This sometimes leads to rather complex code for basic operations.

Compilation can be controlled from the BlueSpec GUI or from a command line interface. To obtain a functionally correct design, the designer needs to understand how the compiler actually generates the RTL code from the BSV input. This requirement, in combination with the proprietary design language and the uncommon rules based programming paradigm, results in a steep learning curve.

BlueSpec scans the source code line by line and generates RTL from there. Signal names from our BSV design were preserved in the RTL code. This has a hugely positive impact with respect to the readability and traceability of the generated code.

The BSV environment offers no design exploration capabilities, since the design entry is at hardware level. The main idea behind BlueSpec is that restrictions that the designer normally needs to hand code in RTL are instead automatically and more consistently coded by the BSV compiler.

For verification of our Sobel filter, we had to manually write a BSV testbench. BlueSpec generates the RTL design in Verilog. In most cases we found that the generated design performs as well as hand-written RTL code for area and speed. From our experience, we conclude that BlueSpec is at an intermediate abstraction level between a high level design language (e.g. C) and RTL. Because of this, BlueSpec can handle both data-oriented and control-oriented applications well.

### 5.5 Catapult C

Catapult C is owned by Calypto Design Systems, acquired from Mentor Graphics in August 2011.

Catapult C accepts a large subset of C, C++ and SystemC. Excellent manuals and a book [8] exist on how to write code for HLS in general and for Catapult C in particular. As a consequence, writing an initial version of the source code was easy.

Catapult C offers an excellent interface that guides the designer through the HLS process. Steps include the selection of source files (design and testbench), design setup (target technology, clock rate etc.), design constraints (area and latency, I/O and array mapping, loop optimizations), scheduling and RTL generation. Most options are prefilled with sensible default values, which makes it easy to get started. The designer gets a very clear view on

different aspects of the design, such as I/O's, arrays, loop nests and the generated schedule. Apart from the GUI, the tool can be run from scripts as well.

The GUI gives access to a range of design tuning options. These include the selection of the target technology and clock rate and the selection of interfaces (e.g. buffered or not). Arrays can be mapped onto registers, RAM, ROM or a streaming interface, and further selections can be made for single or dual port memories, data width, interleaving etc. Loops can be unrolled completely or partially, or they can be pipelined with a certain initiation interval.

Memory accesses are not optimized by the tool, array elements that are reused in subsequent iterations are fetched from memory on every use. To enable local buffering, the designer must modify the source code. The performance of the Sobel edge detector design is limited primarily by memory accesses to the source image. In the initial implementation, eight pixels need to be read to produce one output pixel. With a local buffer for 2 pixel lines, only one new input pixel needs to be read to produce an output pixel. This buffer had to be added in the C++ code manually. The testbench that we designed to validate the source code however didn't need any modifications. After this, further tuning could be done from the GUI including: loop pipelining (so a pixel can be produced in each clock cycle) and also partially unrolling of the inner loop combined with wider memories to allow multiple pixels in each clock cycle.

Arbitrary bit width data types are supported through *Algorithmic C* data types. They are contained in a class-based C++ library from Mentor Graphics that provides arbitrary-length integer, fixed-point and complex data types similar to SystemC data types. Source code using these types can be compiled with any C++ compiler (e.g. g++) and identical behavior of these types in hardware and software is guaranteed (including rounding and saturation behavior).

For verification, Catapult C generates testbenches for ModelSim, for cycle accurate as well as RTL and gate level implementations of the design. On top of that, a verification testbench can be generated that combines the original C++ design and testbench with the generated design which applies the same input data to both designs and compares the output. This speeds up verification substantially. Most of the verification errors we have seen were designer mistakes, e.g. incorrect array size. In an exceptional case, a data hazard was introduced by the tool.

A lot of design variants were generated using Catapult C, usually by changing tool options only. The tool has a built-in revision control system and gives the designer an overview of the key performance figures of each version (area estimation, latency and throughput) in the form of tables and Pareto plots. Performance of the resulting design is good and can be tuned to meet the design objectives.

### 5.6 Compaan

The Compaan tool is a relatively new tool, developed by Compaan design. It is designed for Xilinx FPGAs and works together with the Xilinx toolchain. Compaan is not a full HLS tool: it generates the communication infrastructure for an application but not the processing elements themselves.

Compaan uses the Eclipse SDK tool as its user environment. The main idea behind the Compaan tool is the automatic conversion of streaming algorithms to Kahn Process Networks (KPN), which can then easily be mapped on parallel architectures. The tool is mainly focused on streaming applications and implementation of these applications on heterogeneous multicore platforms. The design entry is a C or Matlab description. Using pragmas one

can indicate which parts of the program need to be auto-parallelised. Auto-parallelisation is only possible when the code is written using nested for-loops. After the KPN translation stage, the designer has to manually map each KPN node to a specific resource. This is either a hardware component, or the node can run as software on a hard- or softcore processor. Our Sobel filter only uses the hardware resources.

Compaan will take care of all the communication between the different processes. Data exchange between different nodes of the KPN map is achieved using a FIFO structure. KPN maps have the specific advantages that they are autonomous, deterministic and make use of distributed control and distributed memory. This allows for easy mapping to a multicore platform. The level of parallelism is achieved by controlling the number of nodes and the number of processors. For design exploration, the source code has to be modified to change the number of generated nodes.

For the nodes that run in hardware, a wrapper with execute, read, write and controller unit is instantiated. Only the control architecture is generated, we still need to write the functional IP ourselves, using external tools. This hardware IP can be generated using Xilinx Coregen, hand-written HDL or one of the other high-level synthesis tools. The nodes that run in software can for example use a MicroBlaze or PowerPC processor. Multiple KPN nodes can run on one MicroBlaze, using the Xilinx OS. Based on the mapping, Compaan generates an XPS project in which each node is connected using a Fast Simplex Link (FSL) interface. This is a uni-directional FIFO interface.

The Compaan tool is a promising tool that focuses on the need for increasing parallelism and multicore support, thereby using the full power of the FPGA platform. The designer thus only has to think about the algorithm. However, Compaan only takes care of the communication; the functionality has to be included by the hardware designer.

### 5.7 C-to-Silicon

C-to-Silicon (CtoS) is a recent HLS tool from Cadence. It uses SystemC as its design language. For designs written in C, C++ or TLM 1.0, CtoS will generate a SystemC wrapper. Unfortunately, the wrappers generated by CtoS use SystemC extensions that do not comply with the OSCI SystemC standard. CtoS is mainly oriented towards ASIC design. An FPGA flow (with a number of optimizations turned off) is included for prototyping purposes.

As CtoS accepts ANSI C and C++, writing source code for the Sobel edge detector was easy. The CtoS GUI however is rather difficult to get started with. A few tutorials are included, but they leave a lot of questions unanswered, a.o. about settings that don't even appear in other tools. Also, defaults are often missing, even if there is only one possible choice. This makes the learning curve quite steep.

Arbitrary bit width and fixed-point data types are supported through SystemC data types. The designer can set a number of options and constraints for loop optimization and memory mapping. These options are often hard to understand, which makes design exploration a difficult task. As opposed to most tools, CtoS generates the technology library while building the design, based on RTL libraries and using an RTL synthesis tool (e.g. Cadence RC or Xilinx XST).

As CtoS generates non OSCI compliant SystemC, verification must be done with the Cadence IUS simulator exclusively. CtoS can generate cycle accurate and RTL models with a verification wrapper. Outputs of the source (reference) and generated designs can be compared on a cycle-by-cycle basis. This would mean that automatic verification won't work for designs generated from untimed source code (a typical use case for HLS), or if the number of clock cycles has changed during scheduling.

### 5.8 CyberWorkBench

CyberWorkBench (CWB) is a HLS tool from NEC. C, SystemC and Behavioral Description Language (BDL) are supported. BDL is based on ANSI C, with non-synthesizable constructs excluded, and extended with a number of hardware specific constructs. CWB comes with a GUI, and individual steps (parsing, compilation ...) can be run from the command line as well.

Starting from an ANSI C design, the designer needs to remove any non-synthesizable constructs. The interface may be specified with function parameters or with global variables. BDL is a superset of C that defines a set of integer data types specifying the associated storage as well as the bit width—e.g. `reg` (register) and `mem` (memory). When ANSI C data types are used, CWB optimizes the number of bits. Our design needed only small code changes on the interface to compile with CWB.

CWB supports automatic and manual scheduling. In manual scheduling mode, the designer delimits clock cycles in the source code. In automatic scheduling mode, the designer can still enforce cycle accurate behavior where desired, which means that a mix of untimed and cycle accurate behavior can be specified. A number of optimizations (e.g. unrolling or folding of all loops, introduction interval ...) may be set globally from the GUI. Alternatively, options for individual elements such as loops and variables may be set in the source code using pragmas.

Based on external RTL libraries and industry standard RTL synthesis tools, CWB allows to generate technology libraries for HLS. From the technology library, CWB subsequently generates function and memory libraries for the design. The designer can restrict the number of resources (multipliers, adders ...) to balance design area and latency. CWB also includes a *system explorer* that generates design implementations for various combinations of compiler settings and presents them in an area/latency diagram.

CWB supports verification at different levels: source code, behavioral, cycle accurate and RTL. Several *simulation scenarios* may be generated, with automatically generated or user provided testbenches. The user may need to convert input data into a separate file for each port, or for memories into a C header file. At times it is difficult to understand the numerous tool options. External industry standard simulators may be called from within CWB. Additionally, CWB supports the addition of assertions in the source code and verify these with the formal property checker.

Depending on the license, the resulting design may be written out in VHDL or Verilog. Additionally, CWB can generate scripts for RTL synthesis. In our experiments, we found that the achievable clock rate after RTL synthesis and FPGA place&route was considerably higher than the one targeted during HLS.

### 5.9 DK Design Suite

DK Design Suite is an older HLS tool that was acquired in 2009 by Mentor Graphics. At that time, Mentor announced that it would support its DK customers until they transition to Catapult C, but later Mentor's website told of major feature enhancements.

The design language is Handel-C, which is a subset of C with a number of extensions, e.g. for the definition of *interfaces*, FIFOs and memories, to support arbitrary precision data types and to define parallelism. The proprietary nature of the language with some counter-intuitive constructs and the need for the designer to explicitly specify parallelism make it a non-trivial job even for an experienced C programmer to write efficient code for the Sobel edge detector.

The DK GUI is quite straightforward and has the look-and-feel of a software IDE. As an alternative, a command line interface is also available. There is an extensive online help system that includes a Handel-C manual.

DK is mainly oriented towards FPGAs, and things like component selection and even pin mapping have to be included in the source code. This makes the code harder to port to a different platform. Design space exploration happens by modifying the source code: by adding macros, defining parallelism, and manually optimizing loop nests. Correctness of optimizations needs to be validated by simulation.

The GUI includes a source level simulator for source code verification, where data can conveniently be read from or written to files through interfaces. Validating the generated RTL design was harder, because we had difficulties getting DK to expose the data ports on the top level and we had to build a testbench by ourselves.

DK design suite generates a design in behavioral VHDL or Verilog, or as a mapped design for an FPGA. The generated design passes through RTL synthesis without problems. The designer can tune the latency and throughput reasonably well, albeit with a lot of manual effort.

### 5.10 Impulse CoDeveloper

The Impulse CoDeveloper tool is developed by Impulse Accelerated Technologies. It generates a RTL design starting from C. A wide variety of FPGA platforms are supported: Altera, Nallatech, Pico and Xilinx. Only data flow applications are supported. Impulse CoDeveloper comes with its own IDE, or alternatively Eclipse can be used. To get started, many tutorials are available.

The tool can be used in two ways: *module generation* and *processor acceleration*. Module generation generates a RTL module that can be combined with other IP to build a complete system, whereas Processor Acceleration creates hardware accelerators that can be connected to an embedded FPGA processor. The ImpulseC language is based on ANSI C with some extensions. We found that the design language achieves a medium abstraction level at the behavioral, algorithmic level.

Some changes to the original C code are necessary to implement the Sobel edge detector with Impulse CoDeveloper. The programming model is based on *Communicating Sequential Processes (CSP)*. The source code remains very well readable. There is however no IP available to the designer. Communication between processes is done with streams, registers, signals, shared memory or semaphores. Several libraries are included to create interfaces between the FPGA and the embedded processor of the target platform, and several bus interfaces are available.

Impulse CoDeveloper offers several options for design optimization. The *Application Monitor* provides a visualization for each of the streams. One can see whether some streams are under- or overused and adjust the depth of the streams, which is a very nice feature towards the designer. The *Stage Master Explorer* is used to analyze how effectively the compiler parallelizes the C statements. Using a dataflow graph for each block, every stage of a pipeline is inspected. Based on this the optimal stage delay is determined using pipeline graphs.

Several simulation, verification and debugging tools are included. An interface with ModelSim is provided. A *Desktop Simulation Model* can be used for fast compilation and validation of prototypes. Depending on the license, the *Stage Master Debugger*, a cycle based hardware simulator, may be included.

The synthesis of our design resulted in HDL files with a large number of lines, thus lowering the readability. An inspection of the code revealed that a large amount of the code describes the interfacing.

### 5.11 ROCCC

ROCCC is an open source HLS tool from the University of California at Riverside (UC Riverside). Commercial support is available from Jacquard Computing, Inc. ROCCC uses eclipse (www.eclipse.org) as its design environment. The Eclipse plugin allows the designer to run ROCCC.

The tool uses a very restrictive subset of C as its design language, often making it hard for the designer to implement the desired behavior. For example, if-then-else constructs can only be used to assign two possible values to the same variable. There is also a rather artificial distinction between modules that are sub-blocks of a design, and systems, representing the top level of a design.

Arbitrary bit width integers are supported through `typedef`'s, but simulation doesn't check for overflows. At compile time, ROCCC allows the designer to set a number of design options. Some options are related to the technology (e.g. how long it takes for various arithmetic operations to complete—information that other HLS tools get from a technology library), while others influence the target architecture.

An interesting aspect about ROCCC are its *smart buffers*. ROCCC analyzes what data elements are fetched from memory and, if the same elements are re-used in a subsequent loop iteration, the data will be stored in the circuit rather than re-fetched from memory. This is very relevant as memory bandwith is a major issue in computing systems already, and with increasing parallelism this memory bottleneck becomes worse. This makes ROCCC particularly useful for streaming or sliding window algorithms. For our Sobel edge design, ROCCC will reuse pixel values so that for each new $3 \times 3$ window only one new value is read, even if the code contains 8 array references per iteration. Unfortunately, the implementation of smart buffers in ROCCC doesn't seem to scale well with the buffer size: in our application, the RTL code would explode to 100+ megabytes of VHDL for a 640-pixel wide image.

With version 0.6 of ROCCC, source code verification has improved a lot. Data into and out of modules and systems are now ordinary C function parameters, so the code can be compiled with any C compiler. Verifying the generated RTL design is more difficult: the memory interface for array data has a mix of synchronous and asynchronous behavior and it forces the memory system to buffer any outstanding requests. A purely synchronous memory system will work with the generated design, but the transfer rate is limited to a single word in every two clock cycles.

### 5.12 Synphony C Compiler

Synphony C Compiler is Synopsys' HLS tool, based on the former PICO tool which they acquired from Synfora in 2010. Design languages are ANSI C and C++.

The architecture of the generated design is a *Pipeline of Processing Arrays* (PPA) that may contain individually compiled *Tightly Coupled Accelerator Blocks* (TCAB). Synphony C Compiler accepts a wide range of C and C++ constructs (including some pointer arithmetic), so implementing the Sobel edge detector was easy. The manual explains well how to write efficient code.

Most optimization options are set in the source code by adding pragmas. Exceptions to this include the clock rate, latency constraints and target technology which are set from the

GUI or from the compilation script. Pipelining with a set initiation interval is supported. The tool can generate memory and streaming interfaces.

We found that using default options may not result in a *feasible* design: Synphony added input and output buffers for all design inputs and outputs, including the input and output images. The result was a huge design (area-wise) that mainly consisted of (badly optimized) data buffers. With the correct settings, this issue was resolved.

Verification is automated and happens at different levels: source code (which is taken as a golden reference), after preprocessing, after scheduling, after synthesis, and on the generated design. External HDL simulators may be called from the tool, including those of other vendors. Simulation results are compared with results of the golden reference. These results are the text output of the testbench, and other testbench output may be added for comparison. This was particularly handy for the Sobel edge detector to verify the output bitmap. By default, simulations are run in the background, but interactive simulation is also possible.

Synphony C compiler generates RTL synthesis scripts. After RTL synthesis, the area is about average in comparison with competing tools.

### 5.13 Tool comparison

From the designers' perspective, different features are desired to make a good HLS tool. The source language should at least be compatible with algorithm design languages, and it should be capable of capturing the design at the right abstraction level like untimed for data flow or cycle accurate for control applications. There should be a flat learning curve for both tool usage and design creation. A HLS tool should have extensive design space exploration options, that at the same time impose the least possible source code modifications. The HLS software should provide integration with the downstream design flow, i.e. with verification and RTL synthesis. Finally, the quality of the design generated with HLS should be (at least) comparable with a handcrafted RTL design.

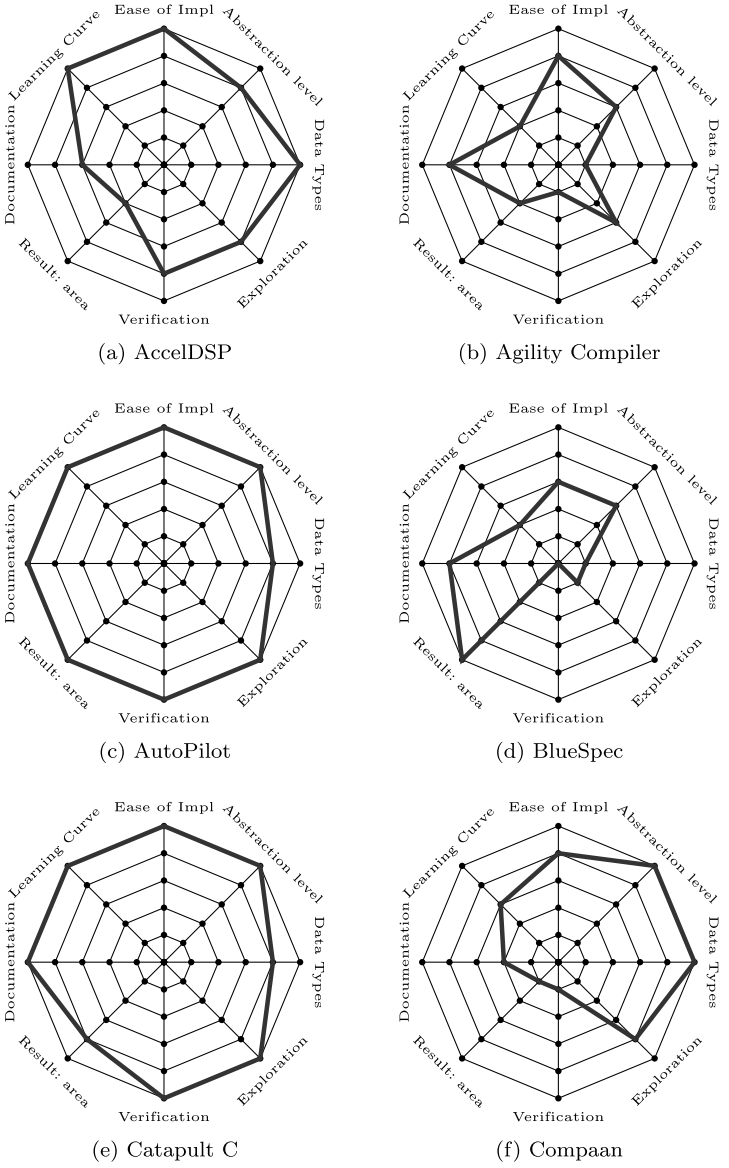
We have presented a large number of commercial HLS tools. A summary of the main characteristics is shown in Table 1. Figs. 4 and 5 represent the same data in a graphical way. The top-left axes of the spider web diagrams represent the user experience, the right hand side and bottom axes highlight the tool capabilities, and the bottom-left axis shows the quality of the result in terms of area. The quality of a tool is better for a certain metric if the spider web is built more to the outside of the corresponding radial axis.

We have extensively discussed scoring criterias within our group, based on our experiences as designers and our feeling on the weight of each criterion. We acknowledge that setting criteria is always somewhat subjective but we believe that this set of criteria and scoring at least provides a good attempt to objectively compare different tools based on different criteria. A designer can still make the choice as to which of the criteria he values more and hence to which part of the spider diagrams he would give more attention. We have set the scoring criteria as follows.

For *ease of implementation*, a 5 means that the original source code could be used with little or no modifications, 3 means that a number of modifications or proprietary language constructs are needed, and 1 means that we had to try several ways to rewrite the algorithm before the tool would accept the code. For *abstraction level*, 5 means that untimed code can be easily scheduled, 4 means scheduling has more restrictions, 3 means that timing is explicit in the high-level source code, and 1 means block level design. For the *learning curve*, scores range from 5 for a flat and 1 for a steep learning curve. This includes how difficult it was to understand source language restrictions, the intuitiveness of the tool, the clarity of

**Table 1** HLS tools summary

| | Source | Abstraction level | Ease of implementation | Learning curve | Documentation | Floating/ fixed pt | Design exploration | Testbench generation | Verification | Size FPGA slices |
|---|---|---|---|---|---|---|---|---|---|---|
| AccelDSP | Matlab | untimed | ++ | ++ | + | auto conversion | + | yes | ++ | 2411 |
| Agility Compiler | SystemC | cycle accurate | + | − | + | fixed pt | + | no | − | |
| AutoPilot | C, C++, SystemC | cycle + untimed | ++ | ++ | ++ | yes | ++ | yes | ++ | 232 |
| BlueSpec | BSV | cycle accurate | +/− | − | + | no | limited | no | − | 120 |
| Catapult C | C, C++, SystemC | cycle + untimed | ++ | ++ | ++ | fixed pt | ++ | yes | ++ | 370/560 |
| Compaan | C, Matlab | untimed | + | + | − | automatic | ++ | no | + | 3398 |
| C to Silicon | SystemC, TLM, C | cycle + untimed | + | − | + | fixed pt | + | yes | + | 1776 |
| CyberWorkBench | SystemC, C, BDL | cycle + untimed | + | + | + | yes | ++ | yes | + | 243/292 |
| DK Design Suite | HandelC | cycle accurate | +/− | − | +/− | fixed pt | limited | no | +/− | 311/945 |
| Impulse CoDeveloper | ImpulseC | cycle + untimed | − | + | − | fixed pt | + | no | + | 4611 |
| ROCCC | C subset | untimed | − | +/− | + | no | + | no | − | |
| Synphony C | C, C++ | untimed | ++ | + | + | fixed pt | + | yes | ++ | 1047 |



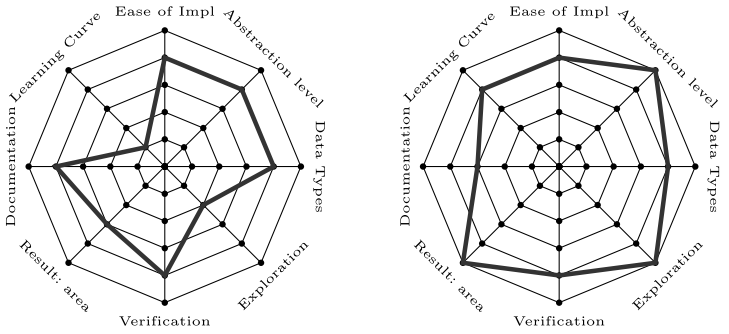

(a) AccelDSP (b) Agility Compiler

(c) AutoPilot (d) BlueSpec

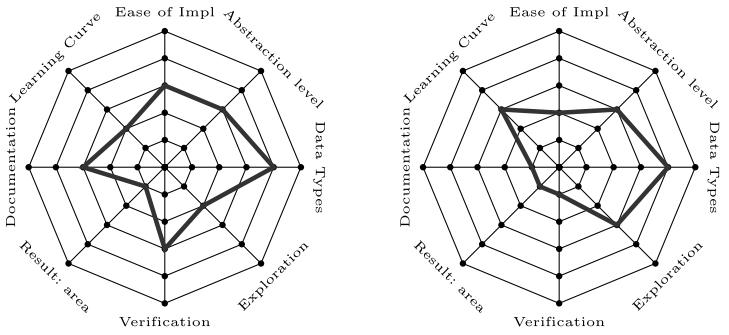
(e) Catapult C (f) Compaan

**Fig. 4** Spider web diagrams for tools 1–6 in the survey

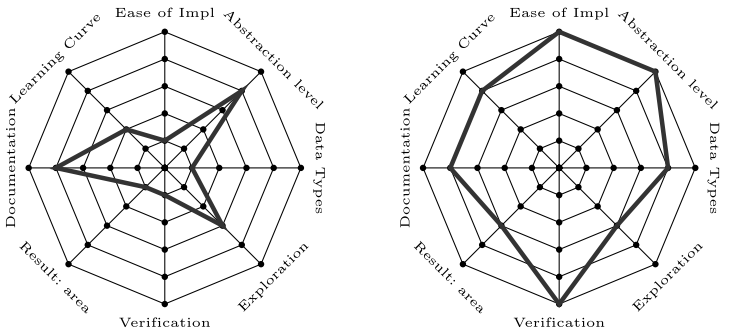
error messages etc. For *documentation*, 5 stands for documentation that is extensive, easy to understand and that explains the *tool concepts*, whereas 1 means that the documentation is too limited, hard to understand or does not explain tool usage well. For *data types*, 5 means that both floating and fixed point are supported, 4 is fixed point only, 3 means that the tool has conversion blocks for fixed point types, and 1 means no support for floating or fixed point.




(a) C-to-Silicon (b) CyberWorkBench




(c) DK Design Suite (d) CoDeveloper




(e) ROCCC (f) Synphony C

**Fig. 5** Spider web diagrams for tools 7–12 in the survey

For *exploration*, 5 means extensive intuitive exploration options that can be set from a script or GUI, 4 means that exploration involved more extensive source code modifications or that tool options were non-intuitive, 3 means that exploration could be done by swapping predefined blocks, and tools with very limited exploration capabilities got 1. For *verification*, 5 means that the tool generates easy-to-use testbenches from pre-existing high-level (e.g. C) testbenches, a tool that generates harder-to-understand testbenches or testbenches that may require data conversion gets a 4, 3 is for a tool with support for source verification but not

RTL, and 1 means that there is no verification support in the tool. For the *result*, we have chosen the amount of resources (as the number of slices on a Xilinx Virtex2Pro/Virtex5 FPGA) to measure to what extent a *reasonably good* result could be achieved in a limited amount of time. Slices are used in all designs and hence more relevant than e.g. RAMs or special functions. Depending on design choices/constraints the number of slices varies (up to 3× in our work), but the differences between tools are much larger (up to more than 20×). 5 stands for the smallest designs (less than 300 slices), 1 for the largest (more than 3000 slices).

## 6 A larger example: WLAN baseband processor

Apart from implementing the relatively small and simple Sobel edge detector with a large set of tools, we also did a more extensive experiment with HLS implementing a WLAN baseband processor with Catapult C. Our implementation started from existing C code that was heavily optimized for a CGRA [15, 16]. The code contained about 20 computation kernels.

Initial porting of the C code to comply with Catapult's requirements took about 2 hours per kernel on average. An important factor was that some of the CGRA optimizations had to be removed. After this, each module could be compiled to (unoptimized) RTL. Fortunately, a C testbench was available for each kernel, so with the help of Catapult's test bench generator (SCverify) the RTL code could be validated in a matter of a few mouse clicks.

At this point, the latency of each kernel was known, and thus the overall latency of the circuit could be estimated. Because this latency didn't meet the specification for real-time, we had to optimize the kernels, which was usually done by creating wider memory interfaces (so multiple words could be transferred in each cycle) and/or pipelining or unrolling loops to increase parallelism. Most of these optimizations did not require C code modifications but only setting options in Catapult C.

The hardest part of the job was to do what the HLS tool couldn't do: create and verify the top level design with all the kernels and the buffer memories and interface logic between them. The complete design turned out to be too large for Catapult C to compile in one run. In comparison with the implementation of the kernels, a massive amount of time was spent on verification of the design, to build the testbench and to search for and correct some (non-systematic) bugs in the handcrafted glue logic. There appears to be a tradeoff: creating a transaction level accurate implementation at the algorithmic level (e.g. in SystemC) would take more effort than purely untimed, but it could save effort when integrating separately compiled modules into a larger design. As this was beyond the scope of our research, it was not further investigated.

This design experience taught us a number of things. First of all, HLS saves the designer a lot of time because existing C code, created during algorithm design, can be used directly to build a circuit. The same applies even more for validation: a module generated with HLS can be validated with a few mouse clicks. On top of that, the number of errors in the generated RTL code is generally low. Most verification errors were caused by incorrect array size specifications in the C code. In such case, bug fixing was a matter of changing one or two lines of C code and re-running synthesis.

Another key strength of HLS is that it allows the designer to explore the design space: try some different constraints and architectures, maybe even take the design through RTL synthesis and place and route, and then pick the most suitable one. Handcrafted RTL design is generally too labor-intensive to allow exploration. Also, a HLS tool has a more global view

on the functionality being implemented and may come up with a globally superior solution, which is similar to modern software compilers outperforming handcrafted assembler.

As we didn't write RTL code by hand we've had to estimate the productivity gain. Based on previous design experience, we believe that analyzing the algorithm and creating an RTL design by hand would take about a week per kernel on average. The creation of the test bench, conversion (where necessary) of test vectors and debugging the design would probably take another 3 days. This has to be compared with about a day when designing with HLS: about 2 hours of porting, and the rest for optimization and verification. This would mean that HLS gives an 8× productivity increase over manual RTL design, which is in line with figures between 2× and 20× that can be found on the Internet.

## 7 Conclusion

In this paper we have presented a number of high-level synthesis tools. These tools raise the abstraction level for designing digital circuits. With these tools, handling the increasing complexity of embedded systems designs becomes easier. HLS tools also take over a number of tasks from the designer and may speed up verification. As such, HLS and FPGAs are a perfect match for high level system implementation, especially when time to market is important.

During our work, we have discovered a number of challenges that HLS tools still face to further improve designer productivity. First of all, standardization in design entry is missing, which means that designers need additional training to design with a specific tool. In all tools, even the best ones, exploration and optimization options could be further extended, e.g. to improve data locality and to reduce memory bandwidth requirements. Often, design space exploration and architecture optimization require source code modifications, which violates the separation between functionality and architecture. Some tools are targeted at a single application domain (data flow or control logic). This too could be a commercial drawback as designers may prefer a single language and toolflow for all their applications.

The above qualitative comparison is meant to provide designers with a good view on current HLS tools. Based on the spider web diagrams, they can decide what tools best perform on the metrics they find most important. For the tool vendors and developers, we hope that this paper will be a stimulus to further improve their tools and as such enhance the possibilities of HLS for everyone's benefit.

**Acknowledgements** This research is supported by the I.W.T. Grant 060068 (SBO) and 80153 (TETRA).

## References

1. BDTi staff (2009) BDTi certified™ results for the Synopsys Synphony C compiler. http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/Synphony. Accessed 30 March 2011
2. BDTi staff (2010) BDTi certified™ results for the AutoESL autopilot high-level synthesis tool. http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot. Accessed 30 March 2011
3. Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z (2011) High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(4):473–491
4. Coussy P, Takach A (2009) Guest editors' introduction: raising the abstraction level of hardware design. IEEE Des Test Comput 26(4):4–6
5. Coussy P, Gajski DD, Meredith M, Takach A (2009) An introduction to high-level synthesis. IEEE Des Test Comput 26(4):8–17
6. DAC (2010) What input-language is the best choice for high level synthesis (HLS)? ACM, New York

7. Duranton M, Yehia S, De Sutter B, De Bosschere K, Cohen A, Falsafi B, Gaydadjiev G, Katevenis M, Maebe J, Munk H, Navarro N, Ramirez A, Temam O, Valero M (2009) The HiPEAC vision: high performance and embedded architectures and compilation. Technical Report 2, July 2009. Deliverable 3.5 of the HiPEAC NoE, version 2
8. Fingeroff M (2010) High-level synthesis blue book. Xlibris, Philadelphia. ISBN 13 (HB): 978-1-4500-9724-6. ISBN 13 (eBook): 978-1-4500-9725-3
9. Gajski DD, Kuhn RH (1983) New VLSI tools. Computer 16(12):11–14
10. Gerstlauer A, Haubelt C, Pimentel AD, Stefanov TP, Gajski DD, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530
11. Green B (2002) Edge detection tutorial. http://www.pages.drexel.edu/~weg22/edge.html. Accessed 30 March 2011
12. Hammami O, Wang Z, Fresse V, Houzet D (2008) A case study: quantitative evaluation of c-based high-level synthesis systems. EURASIP J Embed Syst 2008:685128
13. Martin G, Smith G (2009) High-level synthesis: past, present, and future. IEEE Des Test Comput 26(4):18–25
14. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38(8):144–144116
15. Palkovic M, Cappelle H, Glassee M, Bougard B, Van der Perre L (2008) Mapping of 40 MHz MIMO OFDM baseband processing on multi-processor SDR platform. In: IEEE workshop on design and diagnostics of electronic circuits and systems—DDECS
16. Palkovic M, Folens A, Cappelle H, Glassee M, Van der Perre L (2009) Optimization and parallelization of 40 MHz MIMO SDM-OFDM baseband processing to achieve real-time throughput behavior. In: ICT-MobileSummit
17. Sarkar S, Dabral S, Tiwari PK, Mitra RS (2009) Lessons and experiences with high-level synthesis. IEEE Des Test Comput 26(4):34–45