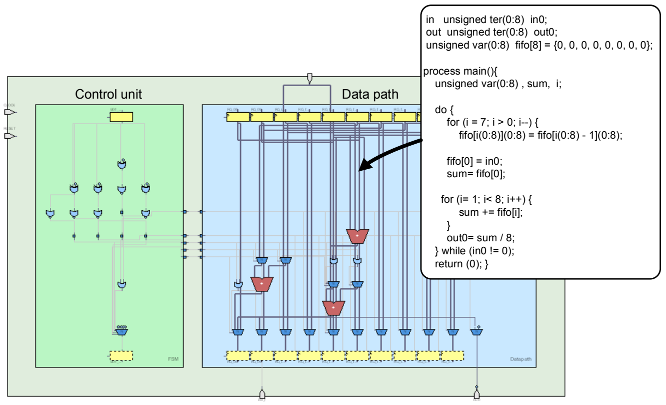
Design of Complex Image Processing Systems in ESL

Benjamin Carrion Schafer[1], Ashish Trambadia[2], Kazutoshi Wakabayashi[3]

NEC Corp. System IP Core Laboratory[1, 3], NECHCL ST[2]
1753, Shimonumabe, Nakahara-Ku, Kanagawa 211-8666 Japan
{schaferb@bq[1], wakaba@bl[3]}.jp.nec.com, ashish.trambadia@nechclst.in[2],

***Abstract*—This work presents the design of a complex image processing IP developed completely in C. We present the latest advanced in ESL-synthesis and demonstrate its main advantages over conventional RT-level flows. In particular we focus on the ability of behavioral synthesis to shorten the design cycle, perform functional verification and explore quickly the design space obtaining multiple dominating implementations with unique area vs. speed characteristics from an initial untimed behavioral description. A feature extraction process is presented in detailed showing how automatic design space exploration can lead to Pareto optimal (non-dominant) designs ranging from 524,648 gates to 584,868 gates and latencies of 38 to 69 state counts for the smallest and fastest design respectively taking approximately 6.3 hours.**



```
in   unsigned ter(0:8)  in0;
out  unsigned ter(0:8)  out0;
unsigned var(0:8)  fifo[8] = {0, 0, 0, 0, 0, 0, 0, 0};

process main(){
   unsigned var(0:8) , sum,  i;

   do {
      for (i = 7; i > 0; i--) {
         fifo[i(0:8)](0:8) = fifo[i(0:8) - 1](0:8);

      fifo[0] = in0;
      sum= fifo[0];

     for (i= 1; i< 8; i++) {
         sum += fifo[i];
      }
      out0= sum / 8;
   } while (in0 != 0);
   return (0); }
```

Figure 1 ESL synthesis results example

INTRODUCTION

It has taken behavioral synthesis 3 generations and over one decade to be seriously considered at the commercial level. Increase design complexity is forcing designers to shift their design methodology from RTL to Electronic System Level (ESL). The main driving force behind the adoption of ESL is the level of maturity of the commercial tools and their capability to deal with system level design issues. This is also the main difference between traditional High Level Synthesis (HLS) and new generation ESL design tools. HLS solves the problems of synthesizing single processes and is still a core part of modern ESL, but the main advantage of ESL is its capability to deal with complete systems. These capabilities include: system level simulation model generation, HW/SW co-simulation, bus interface generation and multi-process physical design synthesis to name a few.

Increasing design complexity leads to new challenges which ESL can address more efficiently than lower levels of abstractions. In many cases the design specifications are unstable and any changes in them can lead to different architectural considerations (e.g. on-die memory or external memory, bus hierarchies). At the RT-level this requires major re-designs, while at the behavioral level these changes can be absorbed easier. Another big advantage of raising the level of abstraction over traditional design flows is that it allows software and hardware designers to speak the same language. Applications to be implemented in custom hardware are getting extremely complex and are based on complex mathematical models that in many cases are difficult to understand by the hardware designer. Using the same behavioral description language allows both hardware and software designers to communicate at the same abstraction level using the same language. Some examples of complex applications include dedicated hardware security engines based on complex encryption and decryption algorithms [1]. Moreover these complex algorithms need to be modified by the hardware designer in order to obtain a more efficient architecture. E.g. reduce memory, split memories, eliminate recursion, improve throughput and latency and data types' refinement. These manual modifications can be hard to do and might lead to the implementation of a design with the wrong functionality. Another aspect that is important when verifying the functionality of the design is the creation of the golden test data. For standard applications this data might already exists, but for complex new applications this data might not be available. In this case who will generate this data? The software or the hardware designer? One last advantage of increasing the level of abstraction is the acceleration of time consuming simulations. RTL simulations are slow and do not allow to simulate entire SoCs. Behavioral simulation models accelerated the simulation and permit the simulation of larger designs.

In this work we present the use of ESL to design a face detection application and in particular we focus on the feature extraction process which is part of the custom hardware face detection accelerator IP designed completely in untimed C. We demonstrate how micro-architectural designs space exploration can be easily performed at the behavioral level, which is extremely time consuming at the RT-level and show how ESL helped the implementation of this application more efficiently.

ESL DESIGN PLATFORM

For the design of the face detection application we used an ESL tool developed in-house at NEC's Central Research Laboratories for the last 20 years called CyberWorkBench (CWB) [2-3]. CWB takes untimed C or SystemC and generates Verilog or VHDL in the traditional HLS way, by creating an FSM for the control unit and a datapath unit. The datapath unit consists mainly of a number of functional units (FUs) combined with registers and multiplexers as shown in Fig. 1. The main idea behind CWB is an "all-in-C" approach. This is built around two principal ideas: (1) All-modules-in-C, and (2.) all-processes-on-C. All-modules-in-C means that all modules in a VLSI design, including control intensive circuits and data dominant circuits, should be described in behavioral C language. CWB also supports legacy RTL or gatenetlist blocks as black boxes, which are called as C functions. At the same time it allows designers to create all new parts in C, although this is not recommended as the designer will need to use two different programming languages and RTL parts will slow down the simulation. All-processes-on-C means

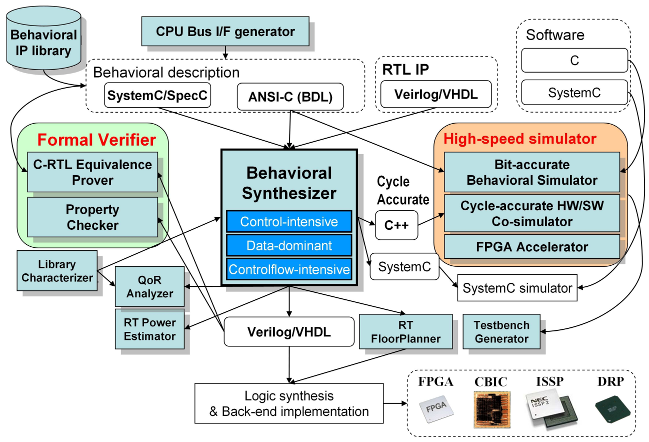


Figure 2 CWB design platform overview

that synthesis and verification (including debugging) tasks should be done at the C source code for single and multiple processes together.

Fig.2 shows an overview of the CWB entire design platform the input to CWB is a hardware design in extended ANSI-C (C for hardware), or SystemC. This is synthesized into synthesizable RTL with CWB's core behavioral synthesizer engine given a set of design constraints such as clock frequencies, number and kind of functional units and memories. As mentioned before the system can also handle legacy RTL blocks as black boxes. These legacy RTL IPs are read into the system and if necessary can also be fed to the behavioral synthesizer, which can insert extra registers to speed up the original RTL and generate new RTL of smaller delay. It also generates different types of simulation models depending on the accuracy vs. simulation speed required. For cycle accurate verification it generates cycle accurate simulation models in C++ or SystemC and for functional verification it can generated behavioral models in SystemC. The behavioral synthesis can therefore be considered as an RTL, C, C++, and SystemC merging step.

To further increase productivity CWB provides a library of behavioral IPs. This library includes trigonometric functions, floating points units and encryption applications. All IPs are given in C and are highly configurable and optimized to obtain optimal RTL.

Wire delays of global wires between modules need to be analyzed carefully since those delays can be significant when the connected modules are placed far away. To account for this the design platform provides an RTL FloorPlanner that takes the RTL modules generated by the behavioral synthesizer. Accurate timing information is extracted from the floorplanner and fed back to the behavioral synthesizer.

Verification Flow

The functionality of the hardware described in C can be verified at the behavioral level, while performance and timing are verified at the cycle-accurate level (or RTL) through simulation. Debugging the generated RTL is however not an easy task since C variables are shared in a register, and various optimizations are applied. CWB therefore provides a behavioral C source code debugger linked to the cycle-accurate simulation. After verifying each hardware module, the entire SoC is simulated in order to analyze the performance and/or to find inter-modules problems such as low performance through bus collision, or inconsistent bit orders between modules. Since such entire chip performance simulation is extremely slow in RTL-based HW-SW co-simulation, CWB generates cycle accurate C++ simulation models which can run up to hundred times faster than RTL models. The simulator allows designers to simulate and debug both hardware and software at the C source code level at the same time. If any performance problems are found, designers can change the hardware-software partitioning or algorithm directly at the C level, and can then repeat the entire chip simulation. This flow implies a much smaller and therefore faster re-design cycle than in a conventional RTL methodology. The C description is the only initial and final SoC description language of the entire design. This entire chip simulation can be further accelerated using an FPGA emulation board. A testbench generator helps designers to run an RTL simulation with test patterns for behavioral C simulation faster and easier. Its inputs are test patterns for the C simulation and output a Verilog and/or VHDL testbench, which generates stimulus for the RTL simulation. It also creates a script to run commercial simulators to feed the behavioral test patterns and check the equivalence of outputs patterns between the behavioral and RTL simulation.

Another important feature of CWB is the formal verification tool, which is tightly linked to the behavioral synthesizer. With the behavioral synthesis information the formal verification tools can handle larger circuits than usual RTL tools and have C-source level debugging capability even though the model checker works on the generated RTL model. "C-RTL equivalence prover" checks the functional equivalence between a behavioral (un-timed or timed) C description and the generated RTL, using information of the optimizations performed such as loop unrolling, loop merge and array expansion performed by the behavioral synthesis. Without such information, the equivalence check is almost impossible for large circuits.

Designers can specify assertions or properties at the behavioral C level, similar to our cycle accurate simulator. Such behavioral level properties/assertions are converted into RTL ones automatically, and are passed to our RTL model checker.

Power Estimation and Minimization

Much work has been done in the past in order to estimate power at different levels of the VLSI design flow ranging from behavioral, RT-level and gatenetlist level Estimation at higher levels of abstraction are needed in order to implement power reduction techniques at the earliest possible design stages avoiding costly re-designs. Moreover powers saving techniques at earlier design stages have larger impact on the power consumption. Fig 3 shows an overview of the main power optimization options at different level of abstraction. The main problem at earlier design stages is that power estimation methods at these stages are not accurate, which can lead

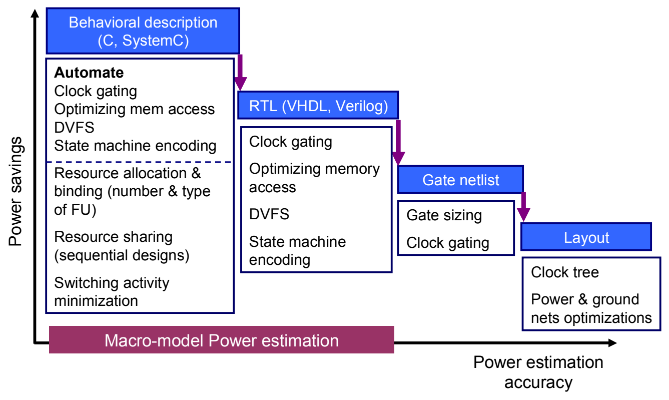

Behavioral description (C, SystemC)
Automate
Clock gating
Optimizing mem access
DVFS
State machine encoding
Resource allocation & binding (number & type of FU)
Resource sharing (sequential designs)
Switching activity minimization
RTL (VHDL, Verilog)
Clock gating
Optimizing memory access
DVFS
State machine encoding
Gate netlist
Gate sizing
Clock gating
Layout
Clock tree
Power & ground nets optimizations
Macro-model Power estimation
Power savings
Power estimation accuracy

Figure 3 Power estimation accuracy vs. power savings options

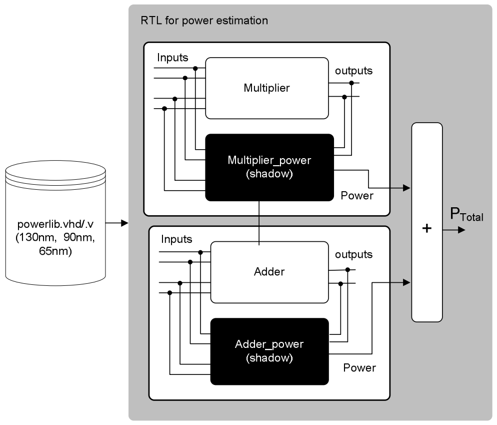
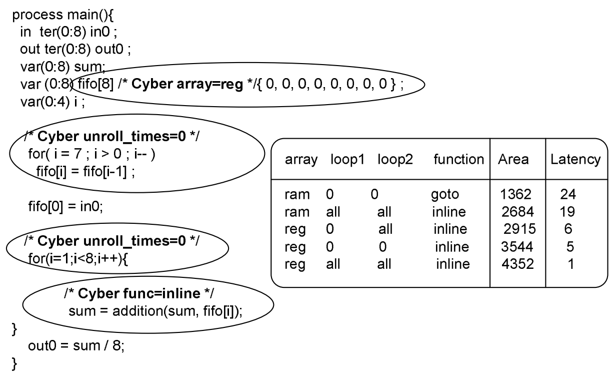


Figure 4 RTL for power estimation



```
process main(){
  in  ter(0:8) in0 ;
  out ter(0:8) out0 ;
  var(0:8) sum;
  var (0:8) fifo[8] /* Cyber array=reg */{ 0, 0, 0, 0, 0, 0, 0, 0 } ;
  var(0:4) i ;

  /* Cyber unroll_times=0 */
   for( i = 7 ; i > 0 ; i-- )
    fifo[i] = fifo[i-1] ;

   fifo[0] = in0;

  /* Cyber unroll_times=0 */
   for(i=1;i<8;i++){

         /* Cyber func=inline */
          sum = addition(sum, fifo[i]);
}
   out0 = sum / 8;
}
```

array	loop1	loop2	function	Area	Latency
ram	0	0	goto	1362	24
ram	all	all	inline	2684	19
reg	0	all	inline	2915	6
reg	0	0	inline	3544	5
reg	all	all	inline	4352	1

Figure 5 Source code example highlighting the explorable operations and summary of HLS results using different set of attributes

to the wrong diagnosis.

In order estimate power, CWB generates apart of the synthesizable RTL, RTL for power estimation. This RTL instantiates shadow components for each atomic unit in the RTL code [4]. These shadow components read the input activity and based on the inputs' values and hamming distance output the power consumption. The power profile of each atomic unit has been pre-characterized in a set of power libraries for different technology [5] as shown in Fig. 4. Based on the power estimation CWB performs different power savings measures e.g. clock gating.

Design Space Exploration

One of the main features of behavioral synthesis is its ability to generated different designs with completely different area vs. performance characteristics. This can be manually performed by manually modifying the initial untimed description or by specifying a number of global synthesis options. CWB also performs design space exploration automatically instrumenting the source code by inserting synthesis directives in the form of pragmas automatically. These pragmas tell the behavioral synthesizer how to synthesize each instrumented operation. *Explorable* operations are operations that have multiple hardware implementations. In behavioral synthesis their implementation is controlled through either (i) global synthesis options or (ii) synthesis directives in the form of pragmas inserted directly at the source code. Global synthesis options have the advantage of being applied at the command prompt and are easy to use. The drawback is the lack of controllability i.e. all the loops will be unrolled if specified as a global options or all functions will be inlined. Pragmas solve this problem by allowing full controllability as they are declared at each operation directly at the source code, but have the drawback that the source code needs to be modified and maintained manually. E.g. loops can be completely unrolled or partially unrolled for lower latency designs. Also arrays can be mapped to registers, memory or hard wired logic for constant arrays. Functions can be inlined which forces the behavioral synthesis tool to instantiate a hardware block whenever the function is called or instantiate a single hardware block sharing it among all the function calls. Fig. 5 shows the source code of a small program that continuously reads in 8 bit numbers and calculates the average of the last 8 values read (same as shown in Fig. 1). The explorable operations have been highlighted and consist of an array where the last 8 numbers are stored, 2 loops and 1 function. The table next to the source code shows the result of the HLS for different synthesis attributes specified directly at the highlighted explorable operations using pragmas. As seen the difference between the smallest but slowest design and the fastest but largest is substantial, ranging from 1362 to 4352 gates and latencies from 24 to 1 cycle. There are a multiple of Pareto optimal combinations in between these designs based on different attribute combinations as well as sub-attributes like the number of memory ports in the array, but only 5 are shown here. Manually editing the source code in order to explore the different area vs. performance trade-offs is a tedious and time consuming task. An automatic efficient design space exploration (DSE) method is therefore highly desirable. The main problem in DSE is how to explore the design space in a reasonable time, finding as many Pareto optimal points as possible.

Parallelization and Pipelining

Pipelining is extremely important to increase throughput and exploit parallelism further. CWB allows pipelining using a pipeline scheduling engine which generates pipelined circuits from the initial C code with stall signals, which have various Data Initial Intervals (DII). It also speeds up loop execution by folding loop and nested loop bodies like software loop pipelining. Global parallelization capabilities are very important even for loop pipelining. Loop carry variables that will be read in the next loop iteration should be scheduled into the states within the given DII cycles sequence. Parallelization beyond control dependencies is one key technique to make loop pipelining possible with a small DII.

FACE DETECTION DESIGN

In this work we present the implementation of a complete face detection design based on Intel's OpenCV library [6] designed and implemented completely in C. Face detection is a process of identifying all image regions with a face regardless of the position in the image [7]. The complete face detection application comprises various processes where the output of each process is passed to the subsequent process. Fig. 6 shows a block diagram of the architectural overview of the system. The system starts by performing a color conversion (rgb2gray) of the image. It then continues by shrinking the image, histogram equalization and sub-image selection. The most computational intensive task of the face detection design is the feature calculation performed next. The face detection process finishes with the face positioning. A control unit synchronizes all the processes working in parallel and makes sure that the all the data is synchronized properly between processes.

When implementing this application the HW designer usually gets the algorithmic description in any high level language e.g. C. This algorithmic description can then be easily converted to hardware, but the designer still needs to have the picture of the overall architecture in mind and manually design the control unit of

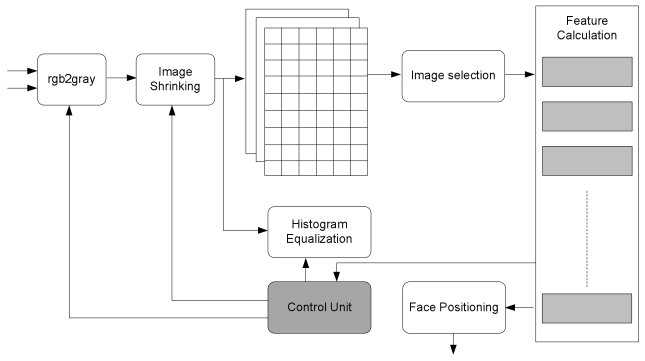


Figure 6 Face detection block diagram

the system. This control unit is again written in untimed C. All these blocks are developed at the behavioral level using untimed C as input language and synthesized with CWB. In this paper focus only on the feature extraction unit as this is the most complex and computational intensive task.

Feature Extraction

The feature extraction process implemented in this work is based on the AdaBoost cascaded algorithm from OpenCV's library [7]. The AdaBoost algorithm uses the boosting classification method of calculating classifiers over the image region and eliminating non-face regions. These classifiers are called HAAR features were each feature consists of a set of black and white rectangles as shown in Fig. 8. The value of each feature is calculated as the weighted sum of the pixels under the white rectangle minus the weighted sum of pixels under black rectangle. A group of different features composes a stage and the stage sum is calculated from the outcome of the features. The result of the stage decides whether the processed region contains a face or not. To speed up the classification process, the original image is transformed to integral and integral squared images. The location of the integral image holds the sum of the intensity values of the pixel located above and to the left of the location in the original image.

The feature calculation unit presented in this work includes all stages mentioned in flow diagram (Fig. 7) based on [7]. The process starts by scanning 20x20 sub windows vertically and horizontally over an original image of a size of 400x300. Scanning is done by shifting 20x20 windows by one row from top to bottom. Once one vertical scan is finished the 20x20 window is shifted by one column from left to right as seen in Fig. 9. At each scanning stage integral and integral squared image values are calculated using following formulas:

$$i(x,y) = \sum O(x',y') \tag{1}$$

$$isq(x,y) = \sum \left[O(x',y')O(x',y') \right] \tag{2}$$

with $x' \leq x, y' \leq y$, where i(x,y) is the integral sum at location (x,y) of the integral image and isq(x, y) is the integral squared sum at location (x,y). O(x',y') is the location on the source image. The Calculated integral sum and integral squared sum values are stored in a 20x20 array matrix which is then used to calculate the pixel sum of the rectangular region of the white and black area. One feature line is calculated using the sum of pixels of two rectangular regions. On the other hand one rectangle sum is calculated with four array references, i.e. one line feature calculation requires eight integral values from the 20x20 array matrix. Similarly, two line

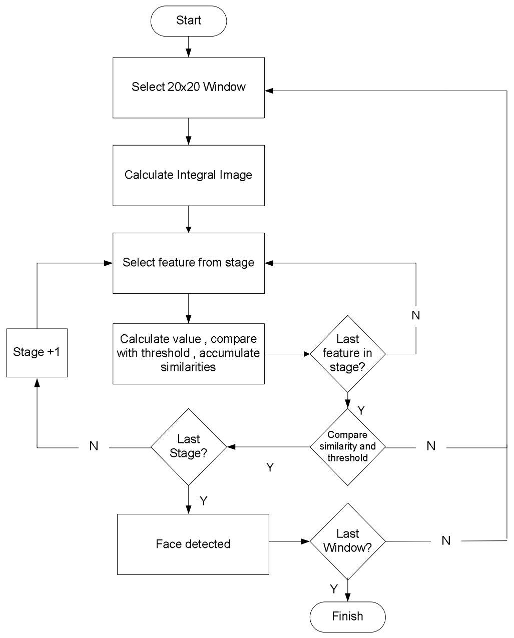


Figure 7 Feature extraction flow graph

features can be calculated using the sum of the pixel of three rectangular regions, thus requiring twelve integral values from the matrix. It seems intuitive that this procedure requires many array references which can easily be parallelized using ESL synthesis by synthesizing arrays registers with the desired number of ports using pragmas inserted directly at the source code as shown:

```
int matrix[512] /* Cyber array=REG, rw_port=RW10 */;
```

This matrix declaration is synthesized as a 32x512 register bank with 10 RW ports making it possible increase data parallelism and therefore reduce latency.

The flow diagram in Fig. 7 shows the selection of the 20x20 sub window for the area to be scanned. Various features are placed on a 20x20 sub window (Fig. 9) one by one and feature value and similarity value of every feature then calculated with the following formula:

$$Fv = w_1 \sum A_T + w_2 \sum A_w \tag{3}$$

Where Fv is the feature value, A_T the total area, A_w the white area and w_1 and w_2 the respective weights

A similarity value of each feature is accumulated for all features in every stage based on it's comparison with a pre-trained feature threshold value stored in an LUT. A window contains a face if a selected sub window passes all the threshold values at all stages. The OpenCV library provides pre-trained data of threshold values for features as well as for stages stored in an LUT to be used during the feature calculation process. Coordinate value (x,y), width, height and weight of every feature are also stored in an LUTs in order to be used during the feature calculation. At the behavioral level these LUTs are

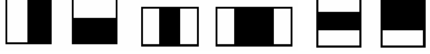


Figure 8 Example of features

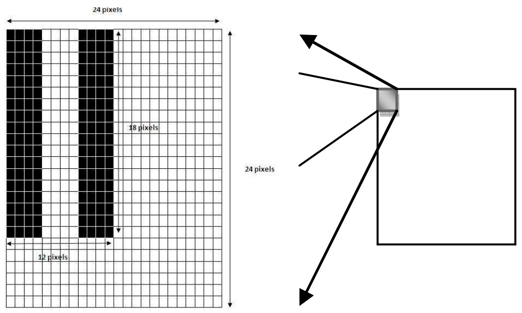


Figure 9 Feature mapping to a 20x20 pixel image

declared as arrays. This leads to one of the major benefits of behavioral level design. Inserting synthesis directives at array declarations changes the way these arrays are synthesized e.g. the features can be stored in a memory, registers, ROM or hardwired logic. The number of read/write ports to access this data can also be modified with a simple synthesis directive allowing the generation of different architecture with the modifying only the synthesis directives.

The complete face detection process performs extensive calculations involving integral image calculation and feature value calculations. Multiple parallel data accesses are also needed to access the pre-trained data stored in the LUTs. The complete process involves different stages, where each stage is composed of many features which make its performance highly dependent on the number of features calculated and how fast each of them can be calculated. To increase the performance it is obvious that multiple features should be calculated in parallel and the integral image calculation should be pipelined to re-use previously calculated data. Parallel processing will require multiple parallel hardware units which will in turn increase the required circuit area. Shi et al. [8] studied this trade-off at the RT-level. In the case of behavioral synthesis this area vs. performance trade-off exploration is done much faster and easier directly from the initial untimed behavioral description. Moreover the same functionality can be described in a behavioral language with less number of lines of code compared to low level RTL descriptions. High level Synthesis tools can generate different architectures of same C description creating e.g. a fully sequential implementation, partial parallel implementation and fully parallel implementation to measure the trade off between area and speed automatically without any manual intervention. This characteristic of HLS proves extremely useful to hardware designers who can the select the design that meets their requirements better.

Experimental Results

This section presents the results of the design space exploration of the feature detection unit. First, we describe the experimental setup for the evaluation. Then, we show a set of comprehensive results obtained, together with the explanation and implication of the analysis of the data.

Experimental Setup

The design was completed written in untimed C. This initial C code description was automatically explored using a previously developed automatic exploration tool [9] in order to investigate the area vs. latency trade-off curve. The number of Pareto optimal designs found, the complete exploration runtime, the number of gates of the smallest design and the latency of the fastest design are investigated. Fig. 10 shows the overall design space exploration flow.

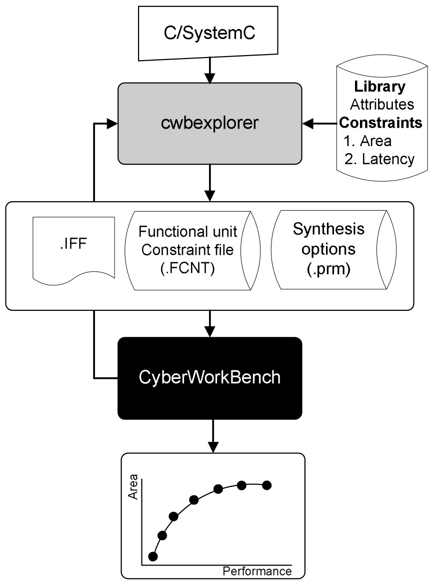


Figure 10 Exploration flow

The exploration tool called cwbexplorer generates for given C or SystemC file a set of attributes that are inserted directly into the parsed C code (.IFF file) a functional unit constraint file specifying the maximum number of FUs allowed and a global synthesis file (.prm) with the commands for the behavioral synthesis. For each unique set of these inputs cwbexplorer calls the behavioral synthesizer and reads back the results in order to evaluate the impact of these con the synthesized results in order to proceed generating a new set of inputs.

The main problem analyzing the results of multi-objective function optimization methods is how to measure the quality of the results. Closeness to the Pareto front, wider range of diverse solutions, or other properties. Several studies can be found in literature that addresses the problem of comparing approximations of the trade-off surface in a quantitative manner. Most popular are unary quality measures, i.e. the measure assigns each approximation set a number that reflects a certain quality aspect, and usually a combination of them is used [12-13]. A multitude of unary indicators exist e.g. hybervolume indicator, average best weight combination, distance from reference set and spacing. Zitzler et al. provide a good review all existing methods in [14]. Although a single unary indicator can not completely measure the quality of the result they are widely used due to their simplicity. In our case we measure the quality of the exploration by comparing the number of non-dominating designs and the smallest and the fastest design. The experiments were run on an Intel Xeon running at 3.20GHz machine with 3Gbytes of RAM running Linux Red Hat 3.4.26.fc3. The running time given comprises the entire exploration process including the behavioral synthesis runtime and the time taken by the cycle accurate simulation for 1,000 inputs.

Experimental Result

Table 1 shows the main characteristics of the feature extraction process. The first column indicates the total number of lines of C code, where the functionality takes 585 lines and the LUTs where the features are stored 2,245 lines. The second column indicates the exploration runtime The third column shows the number of Pareto or at least non-dominating designs found. The last two colums indicate the smallest and the fastes design found.

Table 1 Design space exploration results of feature detection process

# lines of C	# explorable opr.	# Pareto-optimal designs	Runtime [s]	State count fastest	Area smallest [gates]
585 2,245	22	6	22,845	38	524,648

Table 2 shows the detailed exploration result of each of the non-dominating designs in terms of their area, control unit state count, and critical path delay using a 130nm design process library. The control unit state count is used to measure the performance of the design as the process has no un-bounded loop and the latency is equivalent to the state count. In some other cases a cycle accurate simulation is needed in order to obtain the true latency. The drawback is the increase in runtime to perform the exploration. The complete exploration took 22,845 seconds (~6.3 hours) and found 6 non-dominated designs with state counts raging from 38 states to 49 states with an area ranging from 524,648 to 587,956 gates.

Table 2 Design space exploration results of feature detection process

# design	Area [gates]	State count	Critical Path [ns]
1	524,648	69	9.790
2	525,028	68	9.790
3	559,637	61	8.120
4	562,655	49	11.810
5	584,868	48	10.450
6	587,956	38	11.610

Fig.11 displays the Pareto front obtained from the exploration. The 6 design obtained correspond to a highly parallelized design, designs with an intermediate amount of parallelism and a serial design. A more detailed exploration could find some more points on the efficient frontier at the expense of increasing the runtime substantiatlly. RTL is generated automatically for the selected design complete with its testbench and logic synthesis scripts simplifying the designers work even further. On other major advantages of C-based design is that the test data used during the behavioral level model can be completely re-used at the RT-level togther with the golden models, which is very important in the case of novel applications where goldgen models do not exist.

CONCLUSIONS

In this work we present the complete custom hardware face detection implementation in C and describe in detail how raising the level of abstraction simplifies the design. We focus on the feature extraction unit, which is the most computational intensive process in order to further demonstrate how C-base design makes micro-architectural design space exploration possible from the original untimed behavioral description for a fast and easy evaluation of different implementation, something extremely time consuming at the RT-level. System level design capabilities help the design of complete SoCs helping further the faster adoption of ESL design methodologies. We have presented the design platform and its key features to develop complex SoC. Although the presented design is only a single IP block in the SoC it consists of multiple processes that can be designed and verified directly at the C level. State of the art ESL tools have extended system capabilities allowing the evaluation of not only micro-architectural exploration, but also macro, system-level architectural exploration.

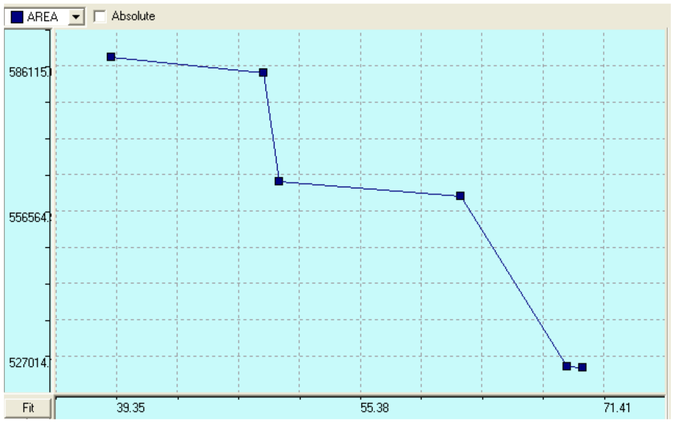


Figure 11 Pareto front results (Area vs. state count)

ACKNOWLEDGMENTS

The authors would like to acknowledge the work of everyone NEC Corp. Central Research Laboratory, NEC Informatec Systems and NEC-HCL-ST for all their work helping the development of CWB and the face detection IP.

REFERENCES

[1] S. Morioka, K. Wakabayashi, B. Carrion Schafer, “Complex Security Engine Design with High Level Synthesis”. MPSoC, 2009.

[2] P. Coussy and A. Moraweic, “High-Level Synthesis from Algorithm Digital Circuit”, Springer, ISBN 978-1-4020-8587-1, pp 113-127, 2008.

[3] CyberWorkBench, “http://www.necst.co.jp/product/cwb/english/”

[4] S Ravi, A Raghunathan and S. Chakfadhar, ``Efficient RTL Power Estimation for Large Designs," International Conference on VLSI Design (VLSI'03), pp.431-440, 2003.

[5] S. Gupta and F. Najm, ``Power Macromodeling for High Level Power Estimation," Proceedings of Design Automation Conference, pp.365-370, Jun. 1997.

[6] Intel Open Source Computer Vision Library, “OpenCV Object Detection: Theory and Practice”

[7] P. Viola and M. Jones, “Robust Real-time Object Detection”, Technical Report CRL 20001/01, Cambridge Research Laboratory, 2001

[8] Y. Shi, F. Zhao and Z. Zhang, “Hardware Implementation of ADABOOST ALGORITHM and Verification”, 22nd International conference on Advanced Information Networking and Applications, 2008

[9] B. Carrion Schafer, T. Takenaka and K. Wakabyashi, “Adaptive Simulated Annealer for High Level Synthesis Design Space Exploration”, VLSI-DAT, 2009

[10] D. Kalyanmoy , S. Agrawal, A. Pratap, and T. Meyarivan, “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II”, Parallel Problem Solving from Nature – PPSN VI, pages 849–858, Berlin, 2000. Springer.

[11] A. David, Van Veldhuizen and Gary B. Lamont. ”On measuring multiobjective evolutionary algorithm performance”, Congress on Evolutionary Computation (CEC 2000), volume 1, pages 204–211,

[11] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. da Fonseca,“Performance assessment of multiobjective optimizers: an analyss and review," IEEE Trans. on Evolutionary Computation, Vol. 7, Issue 2, pp 117-132, 2003.