International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 4 Issue 01,January-2015

Comparative Study of CHStone Benchmarks on Xilinx Vivado High Level Synthesis Tool

Anuj Dubey, Ashish Mishra, Shrikant Bhutada
EEE Group, BITS - Pilani,
Rajasthan, India.

.

Abstract— **High level synthesis is the process of generating hardware for an application written at specification level in languages like C, C++, HDL. Many tools have been developed in the last decade for this process to automates along with numerous optimizations applied is the respective tools. Xilinx also released Vivado, a commercial high level synthesis tool. This paper presents an analysis of the high level synthesis results of CHStone benchmarks using Xilinx Vivado HLS tool and the effect on different optimization on the throughput and area.**

Keywords---High Level Synthesis (HLS), Xilinx, Vivado and CHStone benchmarks.

I. INTRODUCTION

Accelerators are the applications which have been migrated from software to hardware implementation for increasing the performance [1]. In contemporary chips examples of such accelerators are advance encryption algorithm (AES), Cyclic redundancy check (CRC), image processing etc. In such a migration the hardware description (HDL) code is manually written to optimize it for application specific development (ASIC). The performance gain in this process is ten folds but the cost increases because of ASIC based design. The other way around is the automatic generation of HDL and implementation on FPGAs. FPGAs will not deliver as much performance as ASIC based design but chip fabrication is not required. Since migrating a procedural specification to a concurrent design requires efficient bridge for concurrency and timing, the problem has been a popular platform for research in last decade. Many academic and commercial tools have been developed for achieving HLS efficiently. Academic tools include SPARK, ROCCC, LegUp from University of Toronto , GAUT From Universite de Bretagne Sud/Lab-STICC, C-to-Verilog from C-to-Verilog.com and xPilot from University of California, Los Angeles. Commercial tools include BlueSpec Compiler from Bluespec, HLS-QSP from CircuitSutra Technologies, C-to-Silicon from Cadence Design Systems, Concurrent Acceleration from Concurrent EDA, Synphony C Compiler from Synopsys, PowerOpt from ChipVision, Cynthesizer from Forte Design Systems, Catapult C from Calypto Design Systems, eXCite from Y Explorations, Xilinx Vivado (formerly AutoPilot from AutoESL)[2]. This paper aims at bringing out the capability of Xilinx Vivado HLS tool and synthesis results of a specific package of benchmarks called CHStone specially designed for testing the performance of different High Level Synthesis compilers [3]. The tool also provides options for carrying out different types of optimizations on the behavioral description before synthesizing it which enables the user to bring the design closer to the given throughput or area specification. To understand the effect of these optimizations some optimizations have been applied on some benchmarks and the results of the optimized design are compared to the previous design. Towards the end of the paper, Vivado results have been compared with the synthesis results of LegUp compiler [6]. Though they have targeted the Altera Cyclone II FPGA which is different from our target which is Xilinx Kintex7 FPGA, but still the latency and frequency can be compared to some extent just to have a fair idea of where idea about where Vivado stands with respect to other compilers already in market.

II. HIGH LEVEL SYNTHESIS

High-level synthesis (HLS) is an automated process of converting any abstract behavioral description to RTL level so that it can be synthesized into digital hardware that implements that behavior successfully. The given high level specification can be in different forms varying from just an algorithmic description to C/ C++/ SystemC (commonly accepted). The high level description is a procedural description completely free from any kind of clock synchronization which is converted to a completely clocked RTL description by the HLS tool. In a typical VLSI design flow. In a typical VLSI design cycle it is the next step after architectural design and logic synthesis follows after this step.

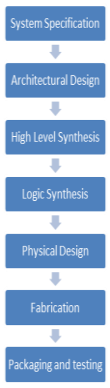

Fig-1 VLSI Design Flow

The tool creates a data flow diagram and control flow graph of the given specification first and then tries scheduling it into different states synchronized to a clock. After this it selects and allocates different hardware resources and functional units to perform the required actions. Here it also attempts sharing wherever possible to minimize resources. After performing all these operations the RTL code is synthesized into digital hardware. The sequence of steps followed in high-level-synthesis is as follows [2]:

1. **Preprocessing**: this process basically refers to the conversion of the algorithmic description into a datapath and a controller. It creates the CDFG (control and data flow graph) of the sequence of events that need to take place to achieve the given functionality.

For e.g.: if the given operation is: y = a * x + b, then its CDFG is:

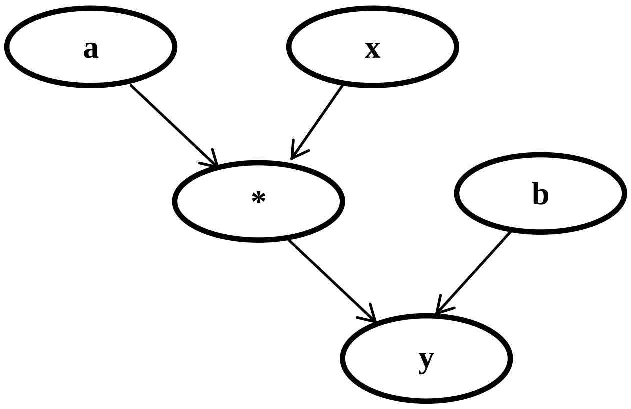

Fig-2 Control and Data Flow Graph

2. **Scheduling**: this is one of the most important steps in HLS where the entire design is synchronized with a clock. It checks dependencies between the different events and their sequence and then schedules them accordingly into different states of a clock.
3. **Allocation**: this step basically deals with the calculation of number of storage elements required for storing the input, output and intermediate values. It also deals with calculation of number of functional units required for carrying out the operations.
4. **Binding**: Variables are mapped to registers, operation to functional units and data transfers to the interconnection units. This step also aims at minimizing the hardware by attempting to share the resources between different units to reduce cost and area.
5. **Data path & Controller design**: controller is designed based on interconnections among the data path elements, data transfer required in different control steps.

III. OPTIMIZATION

There are two important parameters that the designer has to keep in mind while developing any digital hardware viz. area and throughput. The aim of every designer is to minimize area utilization and increase the throughput of the design or in other words optimize the design. These two parameters are inversely proportional to each other and hence both targets cannot be achieved simultaneously. Hence after a particular limit the designer has to make a choice as to what is his/her priority: area or throughput and one have to be compromised in order to achieve the other. There are different kinds of optimizations used by the designer to achieve the functionality according to the required specifications. Some of them are explained below [4]:

1. **Function Inlining**: this basically removes the functional hierarchy which saves the time spent in executing the call and return statements from the function every time it is called. This can be used at places where the function is called just once or twice or if there is some kind of dependency which is preventing the top function to be pipelined.
2. **Function Dataflow Pipelining**: this is a very powerful technique used to increase the throughput by a huge margin. This basically breaks the sequential nature of the algorithm and performs tasks in parallel as much as possible so that one function doesn't have to wait for the previous one to be executed completely before it can start. It checks for dependencies and overlaps the operations as much as possible.
3. **Loop unrolling**: this technique tries to carry out a certain number of iterations of the loop in one go unlike the unrolled case where it executes iteration in each clock. This increases the resources on chip but can prove to be beneficial if the number of iterations is low.
4. **Loop Dataflow Pipelining**: operated in the similar manner as the functions by allowing the parts of the loop that are sequential in nature to occur concurrently at RTL level.
5. **Array Partitioning**: Arrays can also be partitioned into smaller arrays. Memories only have a limited amount of read ports and write ports that can limit the throughput of a load/store intensive algorithm. The bandwidth can sometimes be improved by splitting up the original array (a single memory resource) into multiple smaller arrays (multiple memories), effectively increasing the number of ports.

IV. XILINX VIVADO

The Xilinx® Vivado® High-Level Synthesis (HLS) compiler interface is built very similar to eclipse interface which provides for application development on different types of processors. HLS shares key technology with processor compilers for the interpretation, analysis, and optimization of C/C++ program but the main difference comes in the target execution platform which is an FPGA in our case instead of a processor. This frees the programmer the different constraints in the processor like sharing the same memory source or limited computational resources. We can code assuming that he can take as many resources as possible and also there is no limit on the number of functional units to be used. Also he can use this freedom to guide the design through different kinds of optimizations and bring the design closer to the required latency and area utilization on the FPGA fabric. This becomes particularly very useful for computationally intensive software algorithms (like image processing) which would otherwise take a huge amount of time to execute on a processor. It gives you a proper design metrics of the synthesized design like state table, resources and functional units used by each instance, latency of individual loops etc. which enable the designer to choose which optimization directives should be given to make the design closer to the given specification [5]. We have used Xilinx Vivado HLS version 2013.2 for our purpose, chose the target product family as kintex7_fpv6 and target device as xc7k70tfbg484-2.

V. RESULTS

The synthesis of all the ten CHStone benchmarks was tried using the tool but it found the jpeg benchmark to be non-synthesizable due to the use of dynamic memory allocation in the code. The tool asks for two files viz. the source code file which contains the top level function and a test bench file which basically has the main function and calls the top level function from main. We then check whether the generated output matches with the correct output or not and accordingly return 0 or 1. For example below is the test bench file for adpcm benchmark:

```
#include "adpcm.h"
int main ()
{
FILE *fp;
fp=fopen("out.dat","w");
intretval=0,i;
adpcm_main();
for (i = 0; i< 100 / 2; i++)
 {
        fprintf(fp,"%d",compressed[i]);
 }
for(i=0;i<100;i++)
 {
        fprintf(fp,"%d",result[i]);
 }
fclose(fp);
retval=system("diff --brief -w out.dat out.golden.dat");
if(retval!=0){
printf("Test Failed\n");
retval=1;}
elseprintf("Test Passed\n");
returnretval;
}
```

Here the generated output file is checked with the file containing the correct output and accordingly the return value is decided.

The loop bounds in the C code to be synthesized can either be constants or variable. For certain types of variable loop bounds Vivado can calculate the upper loop bound and give the latency of the design but for some it is unable to do so and hence the results are undefined. Now the tool was not able to give the latency and interval values for most of the synthesized benchmarks because the loop limits were variable and undeterminable. It is able to synthesize the design and generates a state table as well which basically shows the order in which each process will happen but it doesn't give how many cycles each process will last for. Such cases have been mentioned as NA in the table.

The results of the synthesis are given below:

Table-1 Estimated Clock Period

Benchmark	Estimated Clock Period(in ns)
Adpcm	8.64
Aes	8.15
Blowfish	8.51
Dfadd	8.65
Dfdiv	8.64
Dfmul	8.64
Dfsin	8.72
jpeg	8.68
mips	8.01
sha	7.19

Table-2 Latency and Interval

Benchmark	Latency(in clock cycles)		Interval(in clock cycles)	
	min	max	min	max
Adpcm	28254	35654	28255	35655
Aes	NA	NA	NA	NA
Blowfish	2	1442	3	1443
Dfadd	2	8	3	9
Dfdiv	NA	NA	NA	NA
Dfmul	1	14	2	15
Dfsin	NA	NA	NA	NA
jpeg	NA	NA	NA	NA
mips	75	867	76	868
sha	103587	151605	103588	151606

Table-3 Utilization Estimates

Benchmark	BRAM(%)	DSP48(%)	FF(%)	LUT(%)
Adpcm	9	48	5	17
Aes	4	2	12	41
Blowfish	1	~0	1	5
Dfadd	~0	~0	2	26
Dfdiv	~0	10	18	50
Dfmul	~0	6	1	12
Dfsin	1	18	23	93
jpeg	24	38	12	57
Mips	1	3	~0	4
sha	3	~0	1	6

Optimizations: Various optimizations were performed on some of the benchmarks based on the performance and resource profile of the synthesis provided by Vivado. The benchmarks and the optimization directives are given below. The reason why a particular optimization directive is applied on the benchmark and where is it applied is also explained below:

Table-4 list of benchmarks and the optimizations applied

Benchmark	Optimization
adpcm	Function pipelining
blowfish	Array partitioning
dfmul	Function pipelining
mips	Loop unrolling, array partitioning
sha	Function pipelining

- In the **adpcm** benchmark, pipeline directive was applied on the encode function because it contributed maximum latency. This led to a drop in the latency and interval of the design by almost 80%. The time period of each clock cycle remained same.

Table-5 Performance comparison of original and optimized adpcm synthesis

		Solution1	Solution2
Latency	min	28254	7154
	max	35654	7154
Interval	min	28255	7155
	max	35655	7155

Table-6 Resource usage comparison of original and optimized adpcm synthesis

	Solution1	Solution2
BRAM_18K	26	24
DSP48E	116	242
FF	4577	8500
LUT	7293	11483

- In the **blowfish** benchmark, the array partition directive was applied on the ivec array because it got synthesized into a dual port BRAM which was constraining the number of reads and writes per cycle to two. Hence, complete partitioning of the array led to more number of reads and writes per cycle thus decreasing the overall latency and interval of the design. The time period of each clock cycle remained same.

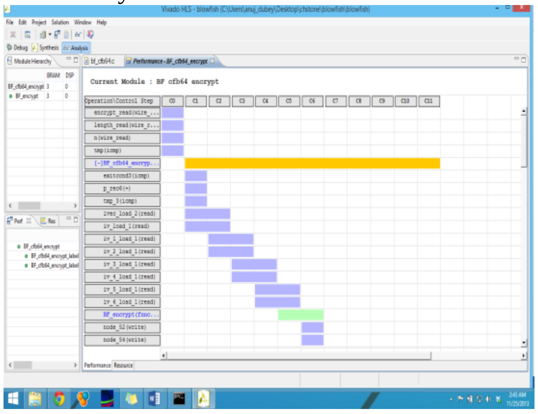

Fig-3 State Diagram before optimization

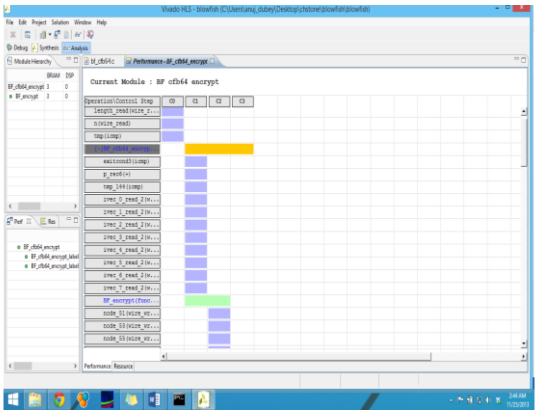

Fig-4 State Diagram after optimization

Table-7 Performance comparison of original and optimized blowfish synthesis

		Solution1	Solution2
Latency	min	2	2
	max	44002	1442
Interval	min	3	3
	max	44003	1443

Table-8 Resource usage comparison of original and optimized blowfish synthesis

	Solution1	Solution2
BRAM_18K	3	3
DSP48E	0	0
FF	1090	1210
LUT	2173	1954

- In the **dfmul** benchmark, the pipeline directive was given to the float64_mul function and it led to a decrease in interval from 14 to 2. The time period of each clock cycle remained same.

Table-9 Performance comparison of original and optimized dfmul synthesis

		Solution1	Solution2
Latency	min	1	14
	max	14	14
Interval	min	2	2
	max	15	2

Table-10 Resource usage comparison of original and optimized dfmul synthesis

	Solution1	Solution2
BRAM_18K	1	1
DSP48E	16	16
FF	1338	1879
LUT	5213	5393

- In **mips** benchmark, the loop unroll directive was given to the three inner loops with constant bounds inside the infinite while loop. This led to a decrease of about 5% in latency and interval. Then further the reg array was completely partitioned. This led to a further decrease of 8% in the overall latency and interval of the design. It also decreased the time period of each clock cycle increasing the frequency.

Table-11 Time period comparison of original and optimized mips synthesis

Clock(ns)		solution1	solution2	solution3
default	Target	10.00	10.00	10.00
	Estimated	8.01	8.01	7.25

Table-12 Performance comparison of original and optimized mips synthesis

		solution1	solution2	solution3
Latency	min	75	27	8
	max	867	819	800
Interval	min	76	28	9
	max	868	820	801

Table-13 Resource usage comparison of original and optimized mips synthesis

	solution1	solution2	solution3
BRAM_18K	4	2	0
DSP48E	8	8	8
FF	437	386	7607
LUT	1900	2120	22748

- In the **sha** benchmark, pipeline directive was applied on sha_transform function since it contributed majorly to the latency. It led to a drastic decrease of 60% in the latency and interval of the design.

Table-14 Performance comparison of original and optimized sha synthesis

	solution1	solution2
Latency	103587	11067
	151605	59085
Interval	103588	11068
	151606	59086

Table-15 Resource usage comparison of original and optimized sha synthesis

	solution1	solution2
BRAM_18K	10	9
DSP48E	0	0
FF	1315	10543
LUT	2619	26709

Table-16 Comparison with LegUP compiler synthesis results:

Benchmark	Latency		Frequency(Mhz)	
	Vivado	LegUp	Vivado	LegUp
Adpcm	7154	10585	115.74	53
Blowfish	1442	196774	117.51	60
DfAdd	8	788	115.61	102
DfDiv	NA	2231	115.74	71
DfMul	14	266	115.74	93
DfSin	NA	63560	114.68	46
JPEG	NA	1362751	115.21	37
MIPS	800	5184	124.84	78
SHA	59085	201746	139.08	58

VI. CONCLUSION

Table-17 Change in latency and resources after optimization

Benchmarks	Decrease in Latency or Interval (%)	Increase in resources(times)	
		Flip-flops	LUTs
Adpcm	80	~2	~1.5
Blowfish	96	Same	~0.9
Dfadd	11	~2	Same
Dfmul	86	~1.4	Same
MIPS	7.8	~17	~12
SHA	61	~8	~10

As we can see from the above table that the throughput can be increased to a considerable extent using the optimizations provided by Vivado, but this increase comes at the cost of area utilization. Xilinx Vivado ensures minimum increase in area as can be seen in the blowfish, dfadd and dfmul results.

VII. FUTURE WORK

The reconfigurable computing systems require the designer to choose which part of the application should run on hardware and which part on software. For that decision the performance of the application on both software and hardware must be calculated. The trade-off between area and performance on chip becomes the quiescent point of the application. This paper provides the performance results of the applications on hardware. Similar results can be calculated for software as well. These results can then be used by the user to efficiently partition the application into hardware and software components resulting in an optimized and efficient performance.

REFERENCES

[1] Christophe Bobda, Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications,Springer, 2007.
[2] http://en.wikipedia.org/wiki/High-level_synthesis last accessed on November, 2013.
[3] Y. Hara, H. Tomiyama, S. Honda, H. Takada, K. Ishii, CHStone: A benchmark program suite for practical C-based high-level synthesis, IEEE International Symposium on Circuits and Systems, ISCAS 2008.
[4] Vivado Design Suite User Guide, UG902 (v2013.2) July 19, 2013.
[5] Vivado Design Suite Tutorial, UG871 (v2013.2) July 19, 2013.
[6] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, Jason Anderson, The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs, IEEE Int'l Symposium on Field-Programmable Custom Computing Machines (FCCM), Seattle, WA, May 2013.