Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 2, Number 1, pages 12–17, January 2013

An Automatic Code Modification and Optimization System for High-Level Synthesis

Mao Hatto †, Takaaki Miyajima ‡, Takaya Toda ‡, Hideharu Amano †
† *Graduate School of Science and Technology, Keio University, Japan*
asap@am.ics.keio.ac.jp

***Abstract*—FPGA (Field Programmable Gate Array) has been used to recent studies and commercial products in high performance computation systems. Hardware Description Language(HDL) for FPGA is sometimes difficult for software engineers who do not have experience and knowledge of hardware design. Solving these issues, High-level Synthesis(HLS) Language, which generates HDL from C-based language automatically, has gathered attention recently. Most HLS provides compiler directives, a combination of sentences and numbers, to search better trade-off. By using this, trial-and-error process to achieve desire trade-off has became much earsier than HDL. Nevertheless to understand relationship between the directives and the final result, user are still required HLS depend experience. For this issue, we proposes an *Automatic Code Modification for High-Level Synthesis (ACM-HLS)*, a tool for clarify the above mentioned relationship. In this paper, ACM-HLS focuses on loop optimization which has greate responsibility to speed-area trade-off. For evaluations, we chose four applications, and achieves up to 66.0% speed up, and 52.3% speed up on average. Furthermore, our tool can also shows a tendency of optimization. Hence, users can chose the combination of directives even they do not have knowledge of experience.**

I. Introduction

While most of studies using FPGAs focus on the performance achievement[1], the difficulty of HDL (Hardware Description Language) coding is increasing as the complication and enlargement of applications are getting forward. Also, reducing the debug time such HDL code is an important problem and research topic. To solve such problems, High-Level Synthesis(HLS) has been received an attention. HDL code is automatically generated from C-like language (HLS language) by using high-level synthesis tool. Impulse C[2], CatapultC[3] and CyberWorkBench[4] have been utilized in recent FPGA design with their synthesis environment.

However, in order to optimize the design, such HLS environment still has problems especially for designers who do not have enough knowledge on FPGA hardware. The compiler directives are the easiest way to achieve a desired result, user choose directive and changes the number for that. For example, loop unrolling is the most important technique to search trade-off between processing time and area. User inserts a "Unroll" directive just above the loop in source code to determine the method of unroll, and change degree of unrolling by changing the number for the directive. After many repeats of such manual changing, user will gets desired result. In particular, the relationship between the combination of directives/numbers and the final result are difficult to understand, and this relationship depends on HLS tools.

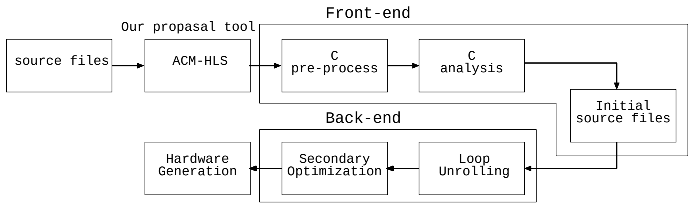

Fig. 1: An synthesis flow of high level synthesis and our tool

We propose and evaluate *Automatic Code Modification for High-Level Synthesis (ACM-HLS)* which is an optimization tool for supporting design exploration using high-level synthesis in order to mitigate the burden of designers. Directives are automatically inserted into the loop structure of the source code and passed to HSL tool. A number of combination of directives are tried and synthesized by using our tools, user just choose hopeful candidates. For optimization, three kinds of unroll methods: loop unrolling, loop partitioning, and loop folding, are automatically controlled. For evaluation, we applied our tool to four applications and obtained tendency of the relationship among directives/numbers.

One of the advantage of our tools is less dependency to HLS environment. This advantage comes from our pre-definition, HLS language is similar to C language, and user just changes the combination of directives and numbers to achieve trade-off. Figure.1 shows outline of common HSL sysnthesis flow. Our tool modiefies inputed source code at the top left. Additionally, CyberWorkBench (CWB) which is a commercial high-level synthesis tool by NEC Corporation, was used as a target synthesis tool.

This paper includes:

- We propose *Automatic Code Modification for High-Level Synthesis (ACM-HLS)*, which is an assist tool for design exploration using high-level synthesis. Directives and are automatically inserted into the loop structure of the source code and the number for directives are also changes to achieve desired result. By using our tool, user efforts are greatly reduced.
- Four applications are applied the tool and three loop unrolling methods: unrolling, partitioning and folding are automatically controlled.

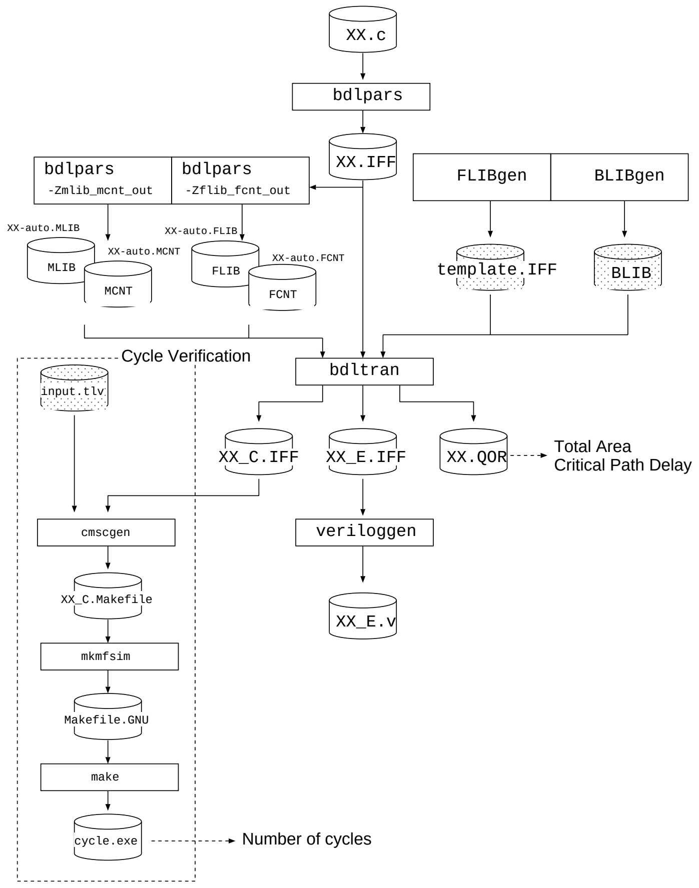

Fig. 2: Synthesis flow in CyberWorkBench

II. An Overview of CyberWorkBench

We firstly explain CyberWorkBench(CWB) as a outline of common high-level synthesis (HLS) flow, and clarify where is ACM-HLS tool. CWB is a commercial tool by NEC, one of the most powerful and clever HLS tool. It automatically generates HDL from a C-like language, called BDL (Behavioral Design Language)[5]. On the other hand, its complexity and the number of options requires user CWB-depend experience and knowledge. We showed outline of common HSL synthesis flow on Figure.1, and this flow is almost the same as that of CWB. From ”C pre-process” to ”C analysis”, it called Front-end that parse inputted source file to a intermediate description. The following processes are called Back-end that generates HDL based on the intermediate description. Our tool modifies source code on the left top of figure, and generates new source code. Many optimization research has been done for the Back-end, but most of them are depends on intermediate language or implementation.

A. Synthesis Flow

Figure.2 shows detail of the synthesis flow of CWB [4]. User write source code in BDL (XX.c), and give it to CWB. The following steps work as follows:

```
/* Cyber unroll_time = all */
for(i=0; i<10; i++){
  //do somthing
}
```

Fig. 3: Usage example of directive

- *bdlpars* converts behavioral description.
  This process corresponds to “C Pre-Processing” for Initial Source files in Figure 1. CWB loads a behavioral description code written in BDL, and converts it into the corresponding internal format (XX.IFF).
- *bdltran* is for behavioral synthesize.
  This process generates a structure-level intermediate file from the format file, which is generated in the previous process. Although this process needs library files and restriction files, CWB can generate them automatically. This process also generates quality file (XX.QOR). We can estimate the total area and critical path delay.
- *veriloggen* generates Verilog-HDL file.
  This process corresponds to the “Hardware generation” in Figure 1. Using the structure-level intermediate file, CWB generates the Verilog-HDL codes.
- *FLIBgen* and *BLIBgen* generate library files.
  These steps create the library file and restriction file. *BLIBgen* generates the basic library, and *FLIBgen* generates the template for restriction file.
- *cmscgen* is for cycle verification.
  This process loads the test data (input.tlv), and executes a cycle verification. We can know required number of cycles via the cycle verification.

B. Directive

CWB has compiler directives, called ”attribute”. This directives are a sentence structured pragma that gives CWB optional information of the circuit generation. Figure 3 shows a usage example of loop unrolling directive. Our tool automatically inserts this directives and changes the option. In addition, this usage is very common in other high-level synthesis tools, such as Impulse C[2]. Some directives have an optional information. In Figure 3, “all” option unrolls the entire of target loop. If a user sets any other number instead of “all” option, the loop is unrolled according to the optional number. More specification of directives will be mentioned in Section IV-B.

III. Related work

Some researches attempted to implement any application on FPGA using high-level synthesis. Jason Cong[1] implements Lithographic Aerial Image Simulation on FPGA. In this research, Impulse C and AutoPilot are used for generating the RTL from ANSI-C code, and focuses on manual code refinements especially for the core nested loop. As the result, up to 15 times speedup over the software implementation was achieved.

Greg Stitt[6] proposes a code refinement methodology for hardware synthesis from C code. He proposes ten methods that improve hardware performance. Using these methods, some hardware/software partitioned applications run 3.5 times faster than the case when partitioning was done on the original unrefined code.

Cwbexplorer [7] has a code modification extension in CyberWorkBench. It is based on an Adaptive Simulated Annealer Exploration Algorithm (ASA-ExpA) for behavioral descriptions written in untimed C or SytemC. Design space exploration by Cwbexplorer is mainly done through attributes that are inserted in the source code. ASA-ExpA generates a new set of attributes for each explorable operation in the source code, based on the previous result, the given area and timing constraints, and a global cost function (GCF). The area and latency weights of the GCF, which represent the importance of either minimizing area or latency, are adaptively modified during the exploration in order to explore the complete area v.s. latency trade-off spectrum. The results of Cwbexplorer show that it successfully searches the design space quickly finding the smallest and fastest design for most benchmarks, incurring in small penalties (5% in area and 8% in latency) for larger benchmarks while reducing the total run-time by an average of 66% compared to a brute force approach. Cwbexplorer has almost the same aims with our research. However, users have to write additional files for exploring the design space. This decreases high usability of high-level synthesis. Furthermore, we tried the direct code modification such as loop partitioning, which does not be implemented in Cwbexplorer.

In general, it is difficult to predict the proper code refinement methods, such as where and which attributes should be inserted. Users have to take many times to know the tendency via such methods. Our tool aimed to show the tendency via code refinement methods, and lighten the burden in high-level synthesis. Furthermore, our research focuses on general high-level synthesis tool, not only for CyberWorkBench.

IV. Overview of the ACM-HLS tool

This section introduces the ACM-HLS tool. Our tool was designed based on the following considerations.

- A target user is an engineer who does not have technical knowledge of hardware synthesis.
- Through the automatic optimization environment, the usability of high-level synthesis tool should be improved.
- Input code is optimized in order to generate the best architecture for user's demand.
- The ACM-HLS shows a tendency of optimization to designers.

A. Automatic Modification and Synthesize flow

Figure 4 shows the overview of ACM-HLS. In CWB, there are some steps for synthesize like Figure 2. User is just required to prepare three information files: information of synthesis environment, test data file and BDL file. Then, ACM-HLS executes the synthesis flow instead of user, and explores the optimized architecture and generates its Verilog-HDL code. ACM-HLS firstly searches the target position to insert directives. All three loop optimization (unrolling, partitioning and folding) methods will be applied to loop block, the tool search "for" blocks. Then, the tool check the iteration times statically.

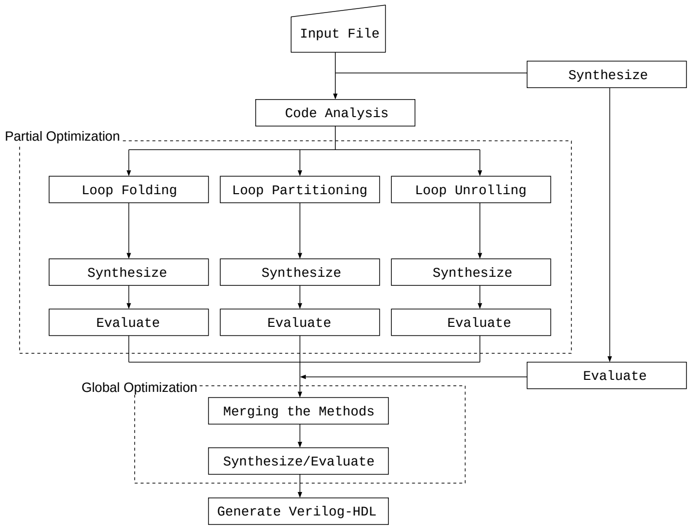

Fig. 4: ACM-HLS synthesis flow

When a loop block was found, ACM-HLS starts optimization flow. The optimization flow is roughly divided into two phases: the partial optimization and the global one. This is because that partial optimum doesn't always equal to global optimum. Unroll less computer intensive loop would be less effective to performance and consume relatively high area. As shown in Figure 4, ACM-HLS applies one of three loop optimization methods to a loop block at the partial optimization phase. Then, ACM-HLS synthesizes and evaluates these optimized codes via CWB. In the global optimization phase, ACM-HLS gathers the best optimization methods of each loop block. Then, it combines these methods and evaluates them. Finally, the results of all generated architectures are sorted, and ACM-HLS generates the closest result to the user's demand.

B. Optimization Methods

In this paprer, we focused on the loop optimization which directly affect to performance and area trade-off. As the methods of loop optimization, we applied three types of unrolling; loop unrolling, loop folding and loop partitioning.

1) Loop Unrolling: Loop unrolling is a common optimization technique to extract parallelism of loop by execution of a number of loops in a single iteration. The derectives for CWB are automatically inserted into the appropriate position around the loop in the source code by ACM-HLS. The synthesis results are controlled by the derectives. For example, by setting the number in the derectives, the degree of loop unroll is defined, except for "all" option which fully unrolls the loop. Figure.5 and 6 show an example of partial loop unrolling two times. ACM-HLS selects the target loop and sets the appropriate options like the example.

2) Loop Folding: Loop folding is an optimization method for loop that pipeline target loop block. This method also

```
for(i=0;i<10;i++){
    x[i] = i;
}
```

Fig. 5: before loop unrolling

```
/*Cyber unroll_time=2*/
for(i=0; i<10; i++){
    x[i] = i;
}
```

Fig. 6: after loop unrolling

```
for(i=0;i<10;i++){
    x[i] = i;
}
```

Fig. 7: before loop folding

```
/*Cyber folding = 1*/
for(i=0; i<10; i++){
    x[i] = i;
}
```

Fig. 8: after loop folding

```
for(i=0; i<10; i++){
    x[i] = i;
}
```

Fig. 9: before loop partitioning

```
for(i=0; i<5; i++){
    x[i] = i;
}
for(i=5; i<10; i++){
    x[i] = i;
}
```

Fig. 10: after loop partitioning

```
for(i=0; i<3; i++){
    x[i] = i;
}
for(i=3; i<6; i++){
    x[i] = i;
}
for(i=6; i<9; i++){
    x[i] = i;
}
for(i=9; i<10; i++){
    x[i] = i;
}
```

Fig. 11: after loop partitioning

improves the performance by parallel processing if possible. Loop folding is done by the directive similar to the loop unrolling. The directive also specifies the number of DII (Data Initiation Interval). ACM-HLS adjusts this option from 1 to available number automatically. Figure.7 and 8 show an example of loop folding with DII = 1.

3) Loop partitioning: Loop partitioning can extract parallelism within loop block. Unlike the previous two optimization methods, this partitioning does not realized by the directive in CWB. ACM-HLS modifies the code to separate the loop directly. Figure 9 and 10 show the example of loop partitioning for dividing a loop with ten iterations into two loops.

In some cases, the number of iteration is indivisible by the number of partition. Fig 11 shows the example of loop partitioning for dividing a loop with ten iterations into three loops. Since ten is indivisible by three, ACM-HLS adds another loop block automatically.

C. Reducing the execution time

As ACM-HLS generates many codes, prosessing time for optimization often becomes longer, few hours or days, and intolerable. Furthermore, if a design includes loops with a lot of iterations, and ACM-HLS unrolls all of them, synthesize time even for single directive position would take a long time. To deal with this problem, We took three ways in order to address this problem.

1) Reducing the number of synthesize patterns:
   ACM-HLS often generates similar patterns and they have the same parallelism. We can thus cut down the pattern those which have the same parallelism as others. More concretely, ACM-HLS sets the optional number of loop unrolling in only divisors of loop iterations. We can use the same technique also in the loop partitioning. As three are the same parallelism in different patterns, it is enough to generate only partitioning in divisors of loop iterations. time.
2) Selecting the target loop:
   Some loop optimizations, for example, initializing the memory do not give any impact to the design performance. Thus, we can reduce the execution time by cutting down optimization for such loops. We weight and rank the operation in each loop. Then, ACM-HLS optimizes a limited number of loops with high priority.
3) Decreasing the synthesis time:
   We tried not to waste generated restriction files. In the optimization flow of ACM-HLS, similar patterns are used and optimized by similar methods. When such patterns are synthesized, the same restriction files are used to save the time.
   Note that, as CWB has an option to reduce synthesis time, we can cut down synthesis time for each trial. However, as this methods depends on CWB, it is not applicable to other high-level synthesis tools. Thus, we did not select this technique.

When using brute force searching, the processing time of ACM-HLS took about a whole day in the worst case. However, it reduced to less than one hour after using these techniques.

V. Evaluation

Four applications, Simple Moving Average, Heap Sort, Lens Distortion Correction and Optical flow, are selected to evaluate ACM-HLS. Four applications include various loop structures, and ACM-HLS explores an optimal directive pattern. In evaluation, we focus on the operating speed and used amount of resource. Additionally, ACM-HLS was implemented using CyberWorkBench version 5.2.2.1, and evaluated targeting with the Virtex-6 series by Xilinx Inc.

In the following result graphs, each name of item is composed in three parts. That is, "optimization methods" "the number of target loop"_"optional number in the optimization

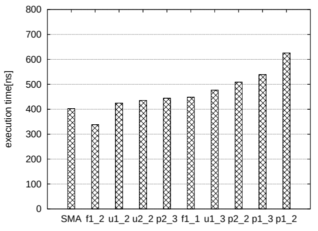

Fig. 12: Execution time (SMA)

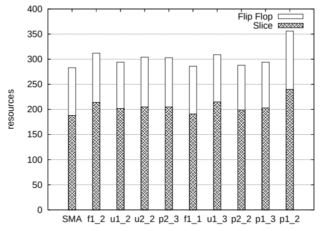

Fig. 13: Amount of resource (SMA)

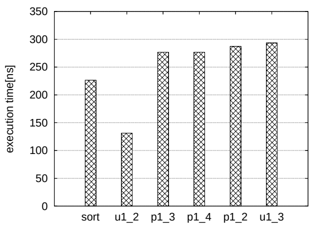

Fig. 14: Execution time (Heap Sort)

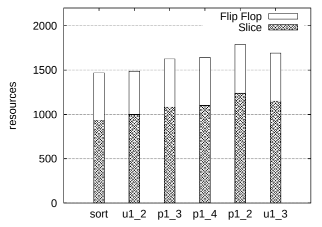

Fig. 15: Amount of resource (Heap Sort)

methods". In the "optimization methods", "u" means "loop unrolling", "f" means "loop folding" and "p" means "loop partitioning". For example, "u1_2" means two times partial loop unrolling for the first loop part.

A. Simple Moving Average

Simple Moving Average (SMA) is small processing that calculates unweighted mean of the previous n data points. In this evaluation, we set n to 8.

Figure 12 and 13 show the results of SMA. The result of the original source code, labeled as "SMA", is at the leftmost in the graphs. In the default settings, CyberWorkBench fully unrolls the target loop blocks. As this function disturbs to find proper optimization methods, we set this function "off". The original source code means a generated results under such CWB setteings, and ACM-HLS also doesn't change anything.

SMA has two loops and their iteration number is seven in this evaluation, ACM-HLS achieved speeding up about 16% via loop folding in DII = 2 for the first loop. Loop folding option pipelines target loop block, and the operational frequency is improved. On the other hand, required resources increased about 10%. Here is a trade-off between execution time and required resources.

B. Heap Sort

Heap Sort is a sorting algorithm that consist of a member of the selection sort family. Double loop in this algorithm, iteration times of inner loop cannot be decided statically. Therefore, only the outer loop became the target of ACM-HLS for optimization. r

Figure 14 and 15 show the results of heap sort. Original source code was labeled as "sort" and located at the leftmost in the graph. ACM-HLS achieved about 42% performance improvement via two times partial loop unrolling for the

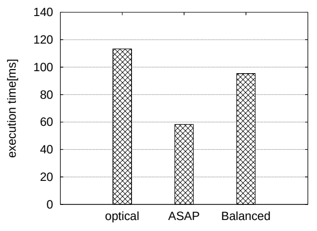

Fig. 16: Execution time (Lens Distortion)

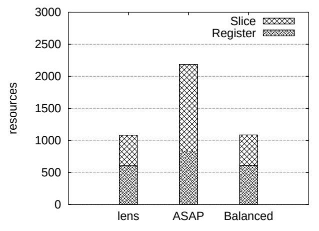

Fig. 17: Amount of resource (Lens Distortion)

TABLE I: Directive Pattern of the partial optimization in the lens distortion

	A	B	C	D	E	F	G	H	I	J	K
p1_32	○	○		○	○		○		○		○
u2_20	○		○	○		○	○			○	○
u3_2		○	○	○				○	○	○	○
u4_2					○	○	○	○	○	○	○

loop. ACM-HLS extracted the instruction-level parallelism, the operational frequency was improved. On the other hand, the increased resource is only about 1.0%.

C. Lens Distortion Correction

Unlike the previous two applications, lens distortion correction is a practical large. Most lenses have bend lines outwards (barrel distortion) or inwards (pincushion distortion)[8]. This algorithm loads an image data that have such distortions, and outputs photo data that have been took away the distortions.

Figure 16 and 17 show the evaluation of lens distortion. Lens distortion has six loop parts, which has a large number of iterations and so the graph shows only three evaluation results: "Lens", "ASAP" and "Balanced". "Lens" is the original directive pattern. "ASAP" achieved the highest operating speed, and "Balanced" is aimed to improved the operational speed with the lowest increasing of resource utilization. In this algorithm, about 50% of speed was achieved, and increase of resource utilization was about 80% in "ASAP" directive pattern. On the other hand, "Balanced" directive pattern had about 16.0% of acceleration and increase of resource utilization was kept in 1.0%. The "ASAP" directive pattern was composed of some optimization methods.

In the step of partial optimization, there were four possible optimization methods to achieve speed-up. Combination of these four methods were 2^4 patterns , and an original source code and four directive patterns that have only one optimization method were synthesized before the partial optimization step. Therefore, the combination of them are $2^4 - 4 - 1 = 11$ patterns. Table I shows those combination patterns, and table II shows the evaluation of each combined optimization method.

According to the Table II, combining all optimization methods (directive pattern K) does not achieve the best result. Besides, there are some directive patterns with performance degrade (directive pattern H and I). ACM-HLS accomplished the evaluation of these combined optimization methods automatically.

TABLE II: Evaluation of whole optimization in the lens distortion

	Execution time[msec]	Slice	Flip Flop	Speedup	decrease of resource
A	58.1	3239	1370	48.7%	-173.9%
B	107.3	1926	699	5.4%	-56.0%
C	56.3	2186	833	50.3%	-79.4%
D	56.3	3784	1429	50.3%	-209.7%
E	107.3	1816	691	5.4%	-49.0%
F	56.3	2218	839	50.3%	-81.6%
G	61.9	4015	1484	45.4%	-226.7%
H	125.2	2676	885	-33.3%	-1.0%
I	125.2	2867	876	-33.3%	-6.2%
J	93.9	2666	860	17.1%	-109.5%
K	62.1	4803	1599	45.2%	-280.4%

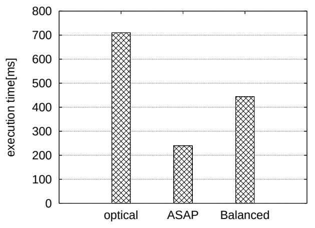

Fig. 18: Execution time (Optical Flow)

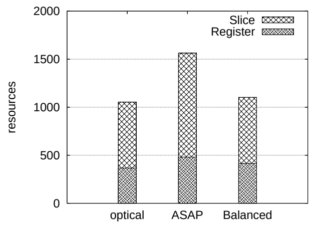

Fig. 19: Amount of resource (Optical Flow)

D. Optical Flow

Optical flow is the pattern of apparent motion of objects in a visual scene utilized in many practical video compressions. From several common optical flow methods, we selected block based optical flow that has a simple algorithm but many iterations. We used a gray scale image data in 200x300 pixels, and set the window size to 3x3. Optical flow also has a double-loop to load image data, and a quadruple loop in the kernel.

Figure 18 and 19 show the evaluation results of the optical flow. In the “ASAP” directive pattern, about 66.0% of acceleration was accomplished, and increase of resource utilization was about 48.0%. On the other hand, “Balanced” directive pattern has about 37.0% of acceleration with only 5.0% increase of resource utilization. In the partial optimization step, applicable methods that increased performance are just for one loop. Therefore, the global optimization step was not executed.

E. Combination with Manual Optimization

Using only automatic optimization, the processing time is reduced by 66.0% at maximum. However, if users step in the flow of optimization easily, more efficient optimization can be possible. At the end of the evaluation, we proposed such collaboration with automatic tool and manual optimization. As this attempt, we chose the application of Lens Distortion Collection. Referring to a code refinement methodology proposed in previous work[6], we modified the code using the following methods: conversion to explicit memory access, elimination of extra branch, and conversion to fixed point computation.

Using the manual optimization, the execution time of Lens Distortion Collection was reduced by 62.6% compared with the original . When we used only ACM-HLS tool, speed-up was 56.3%. It demonstrates that the ACM-HLS is efficient as a basis of manual optimization.

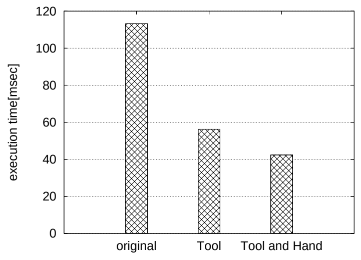

Fig. 20: Execution time with manual optimization

VI. Conclusion

In this paper, we proposed the *Automatic Code Modification for High-Level Synthesis* tool using CyberWorkBench. Loop optimization is usually complex and time consuming task to meet desired result, and the task is not only for the beginners of hardware implementations, but also experts. The main contributions are as follows,

- HLS tools independent, modify original source code
- Loop optimization(Unrolling, folding and partitioning)
- Two step optimization: not local optimum but global one
- Show the tendency of loop optimization of target

We evaluate ACM-HLS by using four applications, the result shows successfully accelerate execution time by 66.0% at maximum, 52.3% on average. On the other hand, there are trade-off between execution time and resource utilization. ACM-HLS supports a trade-off by generating the balanced design and showing the tendency of loop optimization.

Although the current research only focuses on loop optimization, there are other optimization methods for hardware implementation. As future work, we will try to automate such optimization methods, and lighten the burden for users in the high-level synthesis.

Acknowledgments:

This work is supported in part by Kazutoshi Wakabayashi(NEC) and Yamashiroya Atsushi(NEC).

References

[1] Jason Cong and Yi Zou, “Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration,” in *FPGA*, Feb 2008.
[2] Impulse accelerated technologies, “Impulse C,” in *http://www.impulsec.com/*.
[3] Mentor Graphics, “CatapultC Synthesis Overview,” in *http://www.mentor.com/esl/catapult/overview*.
[4] Kazutoshi Wakabayashi, “CyberWorkBench: Integrated Design Environment Based on C-based Behavior Synthesis and Verification,” in *IEEE*, 2005.
[5] NEC, “BDL reference manual version 3.9,” 2011.
[6] Greg Stitt, Frank Vahid, Walid Najjar, “A Code Refinement Methodology for Performance Improved Synthesis from C,” in *ICCAD*, Nov 2006.
[7] Benjamin Scafer, Takashi Takenaka, Kazutoshi Wakabayashi, “Adaptive Simulated Annealer for High Level Synthesi Design Space Exploration,” in *IEEE*, 2009.
[8] DxO Labs, “Photography, Lens distortion,” in *http://www.dxo.com/*.