

# CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis

Yuko Hara<sup>†</sup>   Hiroyuki Tomiyama<sup>†</sup>   Shinya Honda<sup>†</sup>   Hiroaki Takada<sup>†</sup>   Katsuya Ishii<sup>‡</sup>

<sup>†</sup> Graduate School of Information Science, Nagoya University,

Email: {hara, tomiyama, honda, hiro}@ertl.jp,

<sup>‡</sup> Information Technology Center, Nagoya University,

Email: ishii@itc.nagoya-u.ac.jp,

Furo-cho, Chikusa-ku, Nagoya, Aichi 464-8603, Japan

**Abstract**—In general, standard benchmark suites are critically important for researchers to quantitatively evaluate their new ideas and algorithms. This paper presents CHStone, a suite of benchmark programs for C-based high-level synthesis. CHStone consists of a dozen of large, easy-to-use programs written in C, which are selected from various application domains. This paper also presents synthesis results which will be served as a baseline for researchers to compare their new techniques with. In addition, we present a case study on function-level transformation using a program in the CHStone suite.

## I. INTRODUCTION

High-level synthesis (HLS), or behavioral synthesis, is the technology which automatically translates behavioral level design descriptions into register-transfer level (RTL) ones [1]. HLS techniques have been extensively studied for more than two decades, and a number of HLS tools have been developed so far not only in academia but also in industry. Most of commercial HLS tools developed by the middle of 1990's employed HDLs such as VHDL or Verilog-HDL as their input languages. Since the late 1990's, however, C-based programming languages such as ANSI-C, SystemC, and so on, have become popular rather than HDLs [2], [3], [4]. There exist several reasons for this trend, for example: C-based languages facilitate hardware/software codesign since most embedded software is written in C; C-level functional execution is faster than HDL simulation; a large number of existing algorithms are written in C; the number of C programmers is much larger than that of HDL designers.

Standard benchmark suites are critically important for researchers to quantitatively evaluate their new ideas and algorithms. From the late 1980's to the middle 1990's, the HLS research community had made efforts to develop standard benchmark suites for HLS, and as a result, two sets of benchmark designs, High Level Synthesis Workshop 1992 Benchmarks [5] and 1995 High Level Synthesis Design Repository [6], were released from University of California, Irvine, in 1992 and 1995, respectively. The two benchmark suites cover a wide range of application domains from tiny DSP kernels to relatively large microprocessors, and most of the designs are written in VHDL. Indeed, the benchmarks had contributed to the advancement of the HLS technology in 1990's. As mentioned above, however, the language for HLS has shifted from HDLs to C-based ones, and nowadays, to our knowledge, the HDL-based benchmarks are rarely used.

In fact, the repository in [6] includes eight benchmark programs written in C, but most of them are tiny DSP kernel loops which typically consist of less than one hundred lines of C code. They are still useful for studies on loop pipelining and memory access optimization. However, since HLS is expected as a solution for the design productivity crisis, the HLS research community should address synthesis of more complex circuits in order to make HLS a really practical technology. Actually, several recent studies such as [3], [7], [8] have addressed synthesis from large sequential programs consisting of multiple hundreds of lines of C code. The common problem among such researches is the lack of standard benchmark suites. The C programs in [6] are too small, while benchmark pro-

grams which are widely used in the fields of computer architectures and compilers are too huge and complex for hardware synthesis. For example, the SPEC [9], EEMBC [10] and MediaBench [11] suites are not synthesizable even by state-of-the-art HLS tools.

In this paper, we present CHStone, a suite of benchmark programs for C-based HLS. Key features of CHStone are as follows:

- CHStone consists of 12 programs which are selected from various application domains such as arithmetic, media processing, security, microprocessor, and so on.
- The programs in CHStone are relatively large compared with the DSP kernels which have been widely used in the past literature on HLS.
- All the programs in CHStone have been confirmed to be synthesizable by a state-of-the-art HLS tool.
- CHStone is very easy to use since test vectors are self-contained and no external library is necessary.

This paper also presents synthesis results for the CHStone benchmarks using a HLS tool. These results will be served as a baseline with which HLS researchers can compare their new techniques. In addition, we present a case study on function-level transformation using a program in the CHStone suite.

This paper is organized as follows. Section II describes the overview of CHStone and the features of the benchmark programs in CHStone. Section III shows a case study on behavioral-level transformation using CHStone. Section IV concludes this paper with a summary and current status.

## II. THE CHSTONE BENCHMARK SUITE

In this section, we describe a brief overview of CHStone, and then features of individual programs in CHStone are summarized. Finally, some synthesis results are shown.

### A. Overview

The goal of the CHStone benchmark suite is to promote research on C-based HLS by providing researchers a set of benchmark programs which are practically large but still easy to use. We do not intend to evaluate performance of commercial HLS tools. Therefore, we do not strictly define how to use the CHStone suite.

The CHStone suite consists of 12 programs which have been selected from various application domains. CHStone includes four arithmetic programs, four media applications, three cryptography programs, and one processor. It is not guaranteed that the types and numbers of programs in CHStone are sufficient. To our knowledge, however, there exists no HLS benchmark suite which is guaranteed to be sufficient. CHStone is a first step to develop such benchmark suites.

The CHStone benchmark suite is very easy to use mainly because of the following three reasons. First, the CHStone benchmark programs are written in a limited set of the C language. For example, the following data types and constructs, which are not supported by most of existing HLS tools, are not used: floating-point data, composite data types such as *struct*, dynamic memory allocation, and recursion.

TABLE I. OUTLINE OF THE CHSTONE BENCHMARK SUITE

	Design description	Source
DFADD	Double-precision floating-point addition	SoftFloat [12]
DFMUL	Double-precision floating-point multiplication	SoftFloat [12]
DFDIV	Double-precision floating-point division	SoftFloat [12]
DFSIN	Sine function for double-precision floating-point numbers	Authors' group, SoftFloat [12]
MIPS	Simplified MIPS processor	Authors' group
ADPCM	Adaptive differential pulse code modulation decoder and encoder	SNU [13]
GSM	Linear predictive coding analysis of global system for mobile communications	MediaBench [11]
JPEG	JPEG image decompression	Authors' group, The Portable Video Research Group [14]
MOTION	Motion vector decoding of the MPEG-2	MediaBench [11]
AES	Advanced encryption standard	AILab [15]
BLOWFISH	Data encryption standard	MiBench [16]
SHA	Secure hash algorithm	MiBench [16]

TABLE II. CHARACTERISTICS OF CHSTONE BENCHMARK PROGRAMS

	DFADD	DFMUL	DFDIV	DFSIN	MIPS	ADPCM
Representative data type	64-bit integer	64-bit integer	64-bit integer	64-bit integer	32-bit integer	Array of 32-bit integers
Lines of code	494	363	419	789	232	547
Function	17	16	19	31	1	15
Scalar variable	121	91	110	285	30	268
Array variable	4	4	4	3	5	26
Addition/Subtraction	36	28	45	136	17	156
Multiplication		4	8	17	2	69
Division			2	2		2
Comparison	72	34	50	181	12	73
Shift	65	41	56	214	22	81
Logic operation	129	55	65	310	23	24
<i>if</i> statement	78	39	48	192	4	52
<i>switch</i> statement					3	24
<i>while</i> statement			2	3	1	
<i>for</i> statement	1	1	1	1	3	25
<i>goto</i> statement	26	9	11	58		
Test vector length	46	20	22	36	44	100

	GSM	JPEG	MOTION	AES	BLOWFISH	SHA
Representative data type	Array of 16-bit integers	Array of 32-bit integers	3D array of 32-bit integers	Array of 32-bit integers	Array of 8-bit characters	Array of 8-bit characters
Lines of code	380	1,397	441	723	1,413	1,286
Function	12	31	13	11	6	8
Scalar variable	149	393	274	345	110	64
Array variable	10	46	12	11	12	6
Addition/Subtraction	250	1,038	844	510	280	133
Multiplication	53	148		22		
Division		6		12		3
Comparison	109	243	444	48	15	32
Shift	44	293	350	758	159	59
Logic operation	41	132	166	370	370	87
<i>if</i> statement	97	214	351	28	6	3
<i>switch</i> statement		64		10	8	
<i>while</i> statement	1	27	191	5	5	9
<i>for</i> statement	17	90	6	24	5	20
<i>goto</i> statement	30	75	36			
Test vector length	160	7,506	2,048	16	5,200	8,192

Second, the CHStone benchmark programs are written in the standard C language without any extension. Note that most of existing C-based HLS tools extend the C language in order to specify tool-specific optimization options, but the syntax of the extensions are not standardized among the tools. Since such tool-specific extensions are not used in the CHStone benchmark suite, CHStone is highly portable.

Finally, the CHStone benchmark programs are self-contained. Each program has *main* function which serves as a testbench. Test vectors are also contained in the source code. Furthermore, no external library function is called.

### B. CHStone Programs

The benchmark programs in CHStone are brought from widely-used applications in the real world. The programs are selected from various application domains and have different features. Tables I and II summarize the features of the CHStone programs. Table I shows the brief description and the sources of the programs. Table II describes some characteristics of the programs such as the representative data type, the number of lines of C code, the number of functions, the

types and the numbers of operations and statements, and so on<sup>1</sup>. Note that the lines of C code in Table II do not include comment lines or empty lines, so the actual sizes of the source files are much larger. The additional explanation of each program is as follows.

**DFADD:** DFADD implements IEC/IEEE-standard double-precision floating-point addition using 64-bit integer numbers. This program has a number of the control statements such as *if* and *goto* statements. No loop exists except one *for* statement used as a testbench which is added by the authors. This program can be pipelined.

**DFMUL:** DFMUL implements IEC/IEEE-standard double-precision floating-point multiplication using 64-bit integer numbers. This program has several common sub-functions which are also used in DFADD. This program can be pipelined.

**DFDIV:** DFDIV implements IEC/IEEE-standard double-precision floating-point division using 64-bit integer numbers. This program

<sup>1</sup>The numbers of variables, operations and statements described in Table II are based on the case where all the sub-functions are inlined. This is because function inlining is the simplest way to handle global variables. Several parser-level optimizations are done to generate the numbers in Table II.

TABLE III. SYNTHESIS RESULTS FOR CHSTONE BENCHMARK PROGRAMS

	DFADD	DFMUL	DFDIV	DFSIN	MIPS	ADPCM	GSM	JPEG	MOTION	AES	BLOWFISH	SHA
Num. of states	22	24	51	108	18	165	233	815	797	432	350	135
32-bit inc/dec	2				2	9	3	4	4	4	5	3
32-bit add/sub	7	4	4	5	2	7	15	10	8	7	4	7
32-bit multiplier			1			4	1	4		1		
32-bit divider						1		1		1		1
32-bit comparator	8	6	8	11	33	9	4	10	5	5	6	5
32-bit shifter					2	1	2	2	3			
64-bit inc/dec				1								
64-bit add/sub	1	3	3	6								
64-bit multiplier		4	4	4	2							
64-bit divider			1	1								
64-bit comparator	9	5	5	11								
64-bit shifter	8	2	2	8								
ROM (bits)	17,024	12,032	12,416	12,800	1,920	17,664	7,296	65,712	17,600	18,368	116,544	131,296
SRAM (bits)					3,072	9,408	3,408	160,376	16,768	19,584	34,240	3,232

has several common functions with DFADD and DFMUL. DFDIV contains data-dependent loops, which make it difficult to be pipelined. **DFSIN**: DFSIN implements double-precision floating-point *sine* function using 64-bit integer numbers. It calls DFADD, DFMUL and DFDIV, which are also chosen in CHStone.

**MIPS**: This is a simplified MIPS processor which has 30 instructions. A sorting program is served as test vectors.

**ADPCM**: ADPCM (Adaptive Differential Pulse Code Modulation) implements the CCITT G.722 ADPCM algorithm for voice compression. It includes both encoding and decoding functions, which can be pipelined. The two functions can be also used as independent benchmark programs.

**GSM**: This is a program for LPC (Linear Predictive Coding) analysis of GSM (Global System for Mobile Communications), which is a communication protocol for mobile phones. This program implements only lossy sound compression of GSM.

**JPEG**: JPEG (Joint Photographic Experts Group) transforms a JPEG image into a bit-mapped image. This program is mainly composed of three parts: *huffman*, *idct* and *inverse quantization*. An intelligent behavioral synthesis tool may pipeline the three functions. Alternatively, the three functions can be used as individual benchmark programs.

**MOTION**: MOTION decodes a motion vector formatted according to the MPEG-2 standard, which is one of the decompression method of video, audio and so on.

**AES**: AES (Advanced Encryption Standard), also known as Rijndael, is a symmetric key cryptosystem. The AES program includes both *encryption* and *decryption* functions, which can be also used as two benchmark programs.

**BLOWFISH**: BLOWFISH implements a symmetric block cipher. The BLOWFISH program contains only the encryption function.

**SHA**: SHA (Secure Hash Algorithm) is a cryptosystem consisting of a set of hash functions. This SHA program is written so as to conform with Netscapes SSL.

### C. Synthesis Results

As shown in Table II, the CHStone benchmark programs are not small kernel loops but reasonably large applications consisting of multiple hundreds lines of code. We have confirmed that all of the CHStone programs are synthesizable using a state-of-the-art HLS tool. In this section, we present the synthesis results which can be used as a baseline for researchers to compare their new techniques with. In this experiments, we used a commercial HLS tool *xCite* from YXI [17], which takes a C program as input and generates RTL code in Verilog-HDL or VHDL. When synthesizing, no resource constraint is specified, and operation chaining is disabled in order not to depend on a specific target device. No aggressive optimizations were applied such as pipelining, loop unrolling, memory access optimizations and so on. Table III describes the number of states, the types and the numbers of hardware resources, for each benchmark program. For ROM and SRAM, the total numbers of bits are depicted. One may think that the numbers of resources are small, but this is

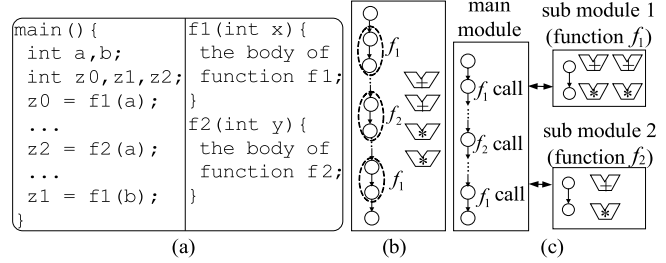


Fig. 1. (a)An example program, (b)The FSM by function inlining, (c)The FSM by function partitioning

because no parallelization techniques were used such as pipelining and loop unrolling. If such optimizations are applied, the designs require more resources and improve performance.

### III. A CASE STUDY USING CHSTONE: FUNCTION-LEVEL TRANSFORMATION

As explained in Section II-A, we expect that the CHStone benchmark suite will be used in researches on HLS. This section presents a case study on HLS where CHStone is used as a benchmark program. Through the case study, we demonstrate the usefulness of CHStone.

The case study focuses on function-level transformation. In general, a large program consists of a set of functions. In the context of C-based HLS, there exist broadly three techniques to handle function calls: *function inlining*, *partitioning* and *goto conversion*. As discussed later, the three techniques have both advantages and disadvantages. In [8], the authors try to establish a systematic methodology for function-level transformation which intelligently combines function inlining and partitioning. In [18], floating-point adders are designed using inlining and goto conversion. To our knowledge, however, no previous work quantitatively compares all of the three techniques. This section first presents an experimental study on function-level transformation where the three techniques are compared.

Function inlining is to replace function calls with the bodies of the called functions. Let us consider an example program shown in Fig. 1(a), where function *f1* requires two adders and two multipliers and function *f2* requires one adder and one multiplier. Fig. 1(b) shows the FSM by function inlining for Fig. 1(a). Note that *f1* is inlined twice since it is called twice. Inlining can keep the total datapath small by sharing the resources among functions. Moreover, there is no performance degradation by inter-module communication. However, the large number of states in main-module may produce an inefficient circuit with a long critical path delay due to the complicated control. This disadvantage becomes significant, especially for programs where large functions are called from many points of the program text.

Function partitioning is to run behavioral synthesis for each function. The FSM by function partitioning for Fig. 1(a) is shown in Fig. 1(c), where one main module and two sub-modules are

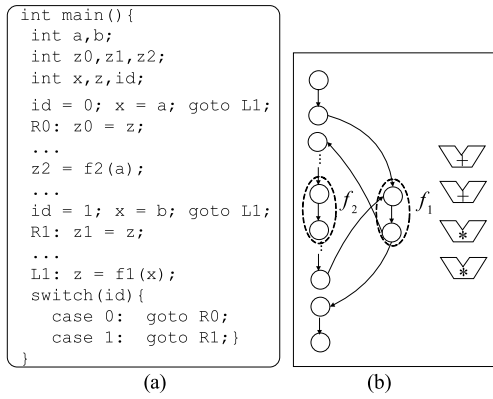


Fig. 2. (a)The rewritten program with goto conversion, (b)The FSM by goto conversion

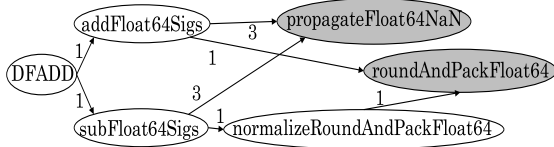


Fig. 3. A call graph of DFADD

generated. Note that only a single module is generated for  $f_1$  even though it is called twice. The main advantage is that it can reduce the complexity of individual modules, resulting in reducing area and delay of the controller circuits. However, the overall datapath area is increased because the resources cannot be shared among functions even though the functions are sequentially executed. Also, performance may degrade by inter-module communication.

Goto conversion is a transformation to realize function calls with goto statements, which is employed in some HLS tools such as [2]. An example of goto conversion for Fig. 1(a) is shown in Fig. 2(a), where only  $f_1$  is goto-converted. The FSM by goto conversion is described in Fig. 2(b), where the body of  $f_1$  is inlined only once in spite of being called at two points of the program text. Moreover, the resources can be shared among functions. However, the state transition may become complex. If all the functions in the program are goto-converted, the main function may become too large.

In these experiments, we select DFADD to evaluate the effectiveness of the function-based synthesis techniques since DFADD has a number of functions which have different features such as sizes and the number of calls. A call graph for DFADD is shown in Fig. 3. The number attached to each arrow represents the number of function calls. In addition to the functions shown in Fig. 3, there exist more than ten functions, but they are omitted here since they are small enough to be inlined. In this case study, we apply three techniques explained above (inlining, partitioning and goto conversion) for each *propagateFloat64NaN* (*prop*) and *roundAndPackFloat64* (*round*) in Fig. 3. The other functions are inlined since they are called only once. Thus, there are totally nine combinations, and for each combination, we performed behavioral synthesis, logic synthesis, and place-and-route to evaluate area and performance of the design. Xilinx Virtex 2 was used as a target device. YXI eXCite was used for behavioral synthesis, and Xilinx ISE was used for logic synthesis and place-and-route [19]. All of these synthesis processes were optimized for performance maximization. Register-transfer level simulation was also performed to measure the execution cycles.

Fig. 4 plots the area-delay trade-offs for DFADD. Each character in the parenthesis represents the applied transformation technique for *prop* and *round*, where I, P and G signify inlining, partitioning and

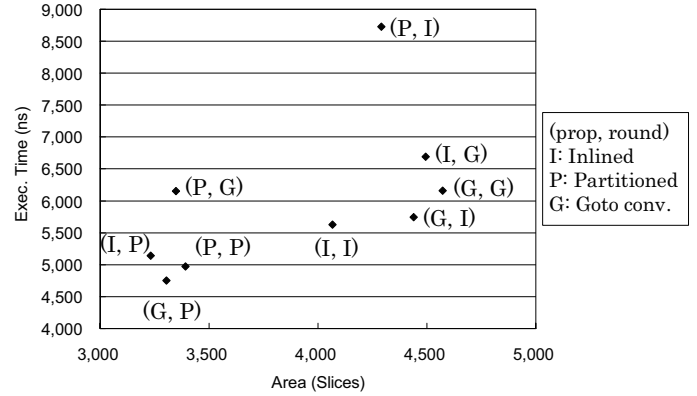


Fig. 4. Area-delay trade-offs for a case study

goto conversion, respectively. Fig. 4 shows that area and performance are significantly affected by function-level transformation. In these experiments, the designs with (I, P) and (G, P) lead to the most efficient results in terms of area and performance, respectively. To our knowledge, however, these optimal designs can not be obtained by previous researches such as [8]. This case study reveals that function-level transformation is still an open problem which needs to be studied using various, large benchmark programs.

#### IV. SUMMARY AND CURRENT STATUS

This paper presented CHStone, a suite of benchmark programs for C-based high-level synthesis. This paper also described synthesis results which will be served as a baseline for researchers to compare their new techniques with. In addition, we presented a case study on function-level transformation using a program in the CHStone suite.

#### ACKNOWLEDGMENTS

This work is in part supported by KAKENHI 19700040.

#### REFERENCES

- [1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
- [2] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Trans. CAD*, vol. 19, no. 12, Dec. 2000.
- [3] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, 2005.
- [4] G. De Micheli, "Hardware synthesis from C/C++ Models," *DATE*, 1999.
- [5] N. D. Dutt and C. Ramchandran, "Benchmarks for the 1992 High Level Synthesis Workshop," *Technical Report 92-107*, University of California, Irvine, 1992.
- [6] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," *JSSS*, 1995.
- [7] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach," *ACM TODAES*, vol. 7, no. 3, July 2002.
- [8] Y. Hara, H. Tomiyama, S. Honda, H. Takada and K. Ishii, "Complexity-constrained partitioning of sequential programs for efficient behavioral synthesis," *GLSVLSI*, 2007.
- [9] SPEC Benchmarks, <http://www.spec.org/>.
- [10] EEMBC, <http://www.eembc.org/>.
- [11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," *MICRO*, 1997.
- [12] SoftFloat, <http://www.jhauser.us/arithmetic/SoftFloat.html>.
- [13] SNU Real-time Benchmarks, <http://archi.snu.ac.kr/realtime/benchmark/>.
- [14] A. C. Hung, "PVRG-JPEG CODEC 1.1," *Technical Report*, Stanford University, 1993.
- [15] AILab, <http://www.ailab.elcom.nitech.ac.jp/>.
- [16] M. R. Guthaus, J. S. Ringenberg, and D. Ernst, "MiBench: A free, commercially representative embedded benchmark suite," *WWC*, 2001.
- [17] Y Explorations, Inc., <http://www.yxi.com/>.
- [18] Y. Hara, H. Tomiyama, S. Honda, H. Takada and K. Ishii, "Behavioral Synthesis of Double-Precision Floating-Point Adecers with Function-Level Transformations: A Case Study," *ICISS*, 2007.
- [19] Xilinx Inc., <http://www.xilinx.com/>.