Journal of Information Processing Vol. 17 242–254 (Oct. 2009)

Regular Paper

Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis

Yuko Hara,[†1,†2] Hiroyuki Tomiyama,[†1]
Shinya Honda[†1] and Hiroaki Takada[†1]

In general, standard benchmark suites are critically important for researchers to quantitatively evaluate their new ideas and algorithms. This paper proposes 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 analyzes the characteristics of the CHStone benchmark programs, which will be valuable for researchers to use CHStone for the evaluation of their new techniques. In addition, we present future challenges to be solved towards the practical high-level synthesis.

1. 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 the commercial HLS tools developed by the middle of the 1990s employed hardware description languages (HDLs) such as VHDL and Verilog-HDL as their input languages. Since the late 1990s, however, C-based programming languages such as ANSI-C and SystemC have become popular rather than HDLs [2)–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; and 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 1980s to the middle of the 1990s, 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 the 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, these benchmarks had contributed to the advancement of the HLS technology in the 1990s. 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 Ref. 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. Several recent studies such as Refs. 3), 7), 8) have addressed synthesis from large sequential programs consisting of multiple hundreds of lines of C code. The common problem shared among such researches is the lack of standard benchmark suites. The C programs in Ref. 6) are too small, while benchmark programs which are widely used in the fields of computer architectures and compilers are too huge and complex for hardware synthesis. For example, C programs in SPEC [9)], EEMBC [10)] and MediaBench [11)] are not synthesizable even by state-of-the-art HLS tools.

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

- CHStone is developed for HLS researchers to analyze the effectiveness of their new techniques, not for LSI designers to evaluate commercial HLS tools.

†1 Graduate School of Information Science, Nagoya University
†2 Research fellow of the Japan Society for the Promotion of Science

© 2009 Information Processing Society of Japan

- CHStone consists of 12 programs which are selected from various application domains such as arithmetic, media processing, and security.
- 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.
- CHStone is available to the public [12].

To the best of our knowledge, CHStone is the first benchmark suite which is composed of practically large programs with various characteristics, refined so that CHStone users can easily and quickly conduct HLS from the CHStone programs, and available to the public.

This paper also analyzes the characteristics of the CHStone benchmark programs, which will be valuable for researchers to use CHStone for the evaluation of their new techniques. In addition, we present some challenges to be solved towards the practical HLS.

This paper is organized as follows. Section 2 describes the overview of CHStone and the brief explanation of each benchmark program in CHStone. Sections 3 and 4 analyze the source-level characteristics of the CHStone programs and sensitivity to resource constraints, respectively. Section 5 discusses the novelty and usefulness of CHStone. Section 6 concludes this paper with a summary and current status.

2. The CHStone Benchmark Suite

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

2.1 Overview

The goal of the CHStone benchmark suite is to promote researches on C-based HLS by providing HLS researchers a set of benchmark programs which are practically large but still easy to use. The past and present HLS researchers have been working towards various goals and metrics (e.g., clock frequency, throughput, area, power/energy consumption, and yield) at various synthesis steps (e.g., source-level transformation, allocation, scheduling, binding, and FSM synthesis). Therefore, we do not define or assume how they use CHStone or what objective they use it for. Also, recall that we do not intend to evaluate the performance of commercial HLS tools with CHStone.

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. Note that many of the CHStone programs are brought from other benchmark suites and are largely changed from their original ones so that the benchmark programs are synthesizable by HLS tools.

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 the existing HLS tools, are not used: floating-point data, composite data types such as *struct*, dynamic memory allocation, and recursive functions.

Second, the CHStone benchmark programs are written in the standard C language without any extensions. Note that most of the existing C-based HLS tools extend the C language in order to specify tool-specific optimization options, but the syntax of the extensions is 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 a *main* function which serves as a testbench. Test vectors are also contained in the source code. Furthermore, no external library function is called.

2.2 CHStone Programs

The benchmark programs in CHStone are brought from widely-used applications in the real world. **Table 1** summarizes the brief descriptions and the sources of the programs. 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. A number of the control statements such as *if* and *goto* statements are used. No loop exists except one *for* state-

© 2009 Information Processing Society of Japan

Table 1 Brief description and source of the CHStone benchmark programs.

Application domain	Name	Description	Source
Arithmetic	DFADD	Double-precision floating-point addition	SoftFloat [13]
	DFDIV	Double-precision floating-point division	SoftFloat [13]
	DFMUL	Double-precision floating-point multiplication	SoftFloat [13]
	DFSIN	*Sine* function for double-precision floating-point numbers	Authors' group, SoftFloat [13]
Microprocessor	MIPS	Simplified MIPS processor	Authors' group
Media processing	ADPCM	Adaptive differential pulse code modulation decoder and encoder	SNU [14]
	GSM	Linear predictive coding analysis of global system for mobile communications	MediaBench [11]
	JPEG	JPEG image decompression	Authors' group, The Portable Video Research Group [15]
	MOTION	Motion vector decoding of the MPEG-2	MediaBench [11]
Security	AES	Advanced encryption standard	AILab [16]
	BLOWFISH	Data encryption standard	MiBench [17]
	SHA	Secure hash algorithm	MiBench [17]

ment used as a testbench which is added by the authors. This program can be pipelined.

DFDIV: DFDIV implements IEC/IEEE-standard double-precision floating-point division using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. DFDIV has several common functions with DFADD. DFDIV contains data-dependent loops, which make it difficult to be pipelined.

DFMUL: DFMUL implements IEC/IEEE-standard double-precision floating-point multiplication using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. No loop exists except one *for* statement, used as a testbench, which is added by the authors. DFMUL has several common sub-functions which are also used in DFADD and DFDIV. This program can be pipelined.

DFSIN: DFSIN implements double-precision floating-point *sine* function using 64-bit integer numbers. A number of the control statements such as *if* and *goto* statements are used. It calls DFADD, DFMUL and DFDIV, which are also included in CHStone.

MIPS: This program describes instruction-level behaviors of a simplified MIPS processor which has 30 types of instructions. A sorting program is served as test vectors. Depending on synthesis options, HLS tools may synthesize a sequential processor or a pipelined one from the program.

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. Only lossy sound compression of GSM is implemented.

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 the Netscape SSL standard.

3. Source-Level Analysis

This section discusses the various features of the programs at the source level. This source-level analysis will be useful for HLS researchers to analyze the effectiveness of their new techniques in experimental results using the CHStone

© 2009 Information Processing Society of Japan

Table 2 Source-level characteristics.

	Representative data type	Lines of C code	Functions	Variables		Operations						Statements					
				Scalar	Array	Addition/Subtraction	Multiplication	Division	Comparison	Shift	Logic	*if*	*switch*	*while*	*for*	*goto/break*	*assignment*
DFADD	64-bit integer	526	17	121	4	38			78	65	146	87			1	26	299
DFDIV	64-bit integer	436	19	111	4	45	8	2	50	56	73	47		2	1	11	220
DFMUL	64-bit integer	376	16	92	4	28	4		34	41	61	38			1	9	159
DFSIN	64-bit integer	755	31	285	3	141	17	2	196	214	357	216		3	1	58	864
MIPS	32-bit integer	232	1	32	5	17	2		12	22	23	3	3	1	3	34	66
ADPCM	Array of 32-bit integers	541	15	269	26	156	69	2	73	81	24	52	24		25	97	792
GSM	Array of 16-bit integers	393	12	150	10	251	53		110	44	41	95		1	17	30	492
JPEG	Array of 32-bit integers	1,692	30	390	48	1,029	148	6	242	277	132	213	64	27	90	228	2,666
MOTION	3D array of 32-bit integers	583	13	146	12	299			155	127	55	115		65	6	12	1,100
AES	Array of 32-bit integers	716	11	345	11	510	22	36	48	758	370	26	10		24	37	909
BLOWFISH	Array of 8-bit characters	1,406	6	112	12	280			15	159	370	5	8	5	5	16	385
SHA	Array of 8-bit characters	1,284	8	66	6	134		3	32	59	87	2		9	20		315

programs. **Table 2** summarizes the source-level characteristics of the CHStone benchmark programs such as the representative data type, the number of lines of C code, the number of functions, and the types and the numbers of operations and statements ⋆1. As seen in Table 2, the representative data types of the CHStone programs are distributed from 8-bit characters to 64-bit integers, and from scalar variables to a three-dimensional array variable. The lines of C code in Table 2 do not include comment lines or empty lines, so the actual sizes of the source files are much larger ⋆2. Thus, we see that the CHStone benchmark programs are realistic applications consisting of multiple hundreds lines of code. Most of the CHStone programs call a lot of functions and form complicated function-call structures such as a function called by multiple functions and the deep function-call hierarchy. Also, the numbers of operations and statements show the complexity of the CHStone programs.

Figures 1 and **2** display the more detailed characteristics of the programs at the source level. Figures 1 and 2 depict the incidence of operations and statements in the programs, respectively.

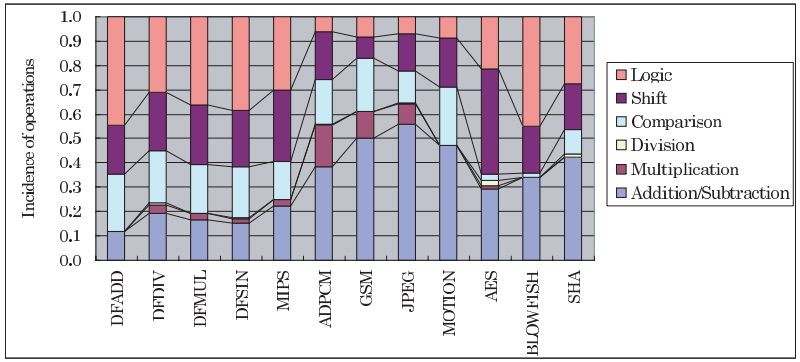

Fig. 1 Incidence of operations per benchmark program.

As stated in the previous section, the CHStone programs are selected from various application domains, and we see from Fig. 1 that each application domain of CHStone has different characteristics in terms of operations. The arithmetic programs have the low proportion of arithmetic operations compared with the programs of media processing and security. This is because the arithmetic pro-

⋆1 The numbers of variables, operations, and statements described in Table 2 are generated after several parser-level optimizations such as function inlining, dead code elimination, constant propagation, and common subexpression elimination.

⋆2 The number of lines in each program is counted after reformatting the program's coding style with GNU indent.

© 2009 Information Processing Society of Japan

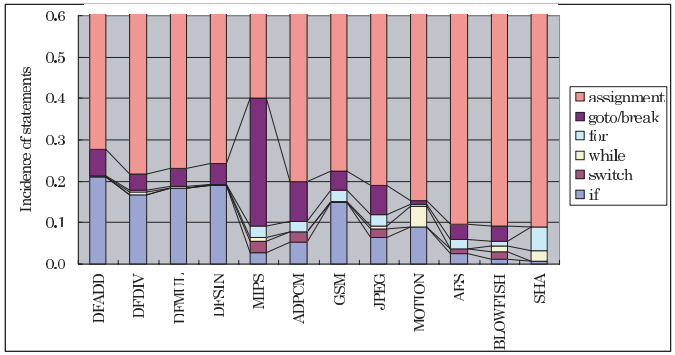

Fig. 2 Incidence of statements per benchmark program.

grams of CHStone deal with 64-bit integer numbers transformed from double-precision floating point numbers and mainly consist of bit operations. Also, the microprocessor application program, MIPS, has the similar feature with the arithmetic programs. This is because MIPS mainly has controls of MIPS instructions rather than operations in an arithmetic logic unit (ALU). The media processing programs, on the other hand, have a high proportion of arithmetic operations, whilst having a low proportion of logic operations, because of the encoding and/or decoding processing to compress or decompress a set of data, such as discrete cosine transform. We see no common characteristics through the security application programs, and the different characteristics among the security programs are observed.

Next, let us look at Fig. 2. Figure 2 shows that each application domain of also has the different characteristics in terms of statements. Except assignment, which is usually dominant in every program, the arithmetic application programs largely consist of condition statements, especially *if* statements. This is because these programs are mainly composed of bit manipulations such as the underflow and overflow handling, as explained in the discussion on Fig. 1. MIPS has the highest proportion of *goto/break* statements due to a few large *switch* statements for 30 types of MIPS instructions. The media processing programs have the high proportion of loop statements since they usually need to repeat the same processing to a set of data. The security application programs have different features on statements as well as explained in the discussion on Fig. 1.

In addition, if we look at Fig. 1 and Fig. 2 more carefully, it can be recognized that the CHStone programs have different characteristics on operations and statements even amongst the programs in the same application domain.

4. Analysis of Synthesis Results

In this section, first the synthesizability of the CHStone benchmark programs is confirmed. Secondly, after we analyze the characteristics in terms of resource utilization, we discuss sensitivity to resource constraints. Finally, future challenges to be solved towards the practical HLS are mentioned.

4.1 Synthesizability

As stated in Section 2, one of the key features in CHStone is that the CHStone benchmark programs are easy to use since CHStone excludes the unsynthesizable data types and constructs by most of the existing C-based HLS tools, such as composite data types, dynamic memory allocations, and recursive functions[⋆1]. In addition, test vectors are self-contained and no external libraries are needed. Also, no tool-specific extensions are used.

Let us discuss the synthesizability of programs in more detail.

The CHStone programs are written in the standard C language. Most of the C-based HLS tools, however, cannot handle the CHStone programs as-is. This is because the tools extend the C language in order to describe some important concepts for hardware design, which are lacking in the standard C language, such as cycle accurate timing for communication and bit-width and direction of I/O ports. The extension varies depending on the tool. We call the extension *tool-specific extension*. If we want to synthesize from a standard C program, we need to modify the program according with the tool-specific extension of the HLS tool to make the program acceptable by the tool.

In this paper, we say that a program is *synthesizable by an HLS tool* if the HLS tool can generate an RTL circuit from the program without modifying the program except for the tool-specific extension. According to the definition, programs

⋆1 Although C-based HLS tools take C-like programs as input, they do not completely comply with the ANSI/ISO C standard. Only a subset of the C language is acceptable by the HLS tools.

© 2009 Information Processing Society of Japan

Table 3 Ratio of the tool-specific lines to the total lines of code.

	Lines of code in CHStone	Lines of code after modification	Ratio of tool-specific extension
DFADD	526	517	4.75%
DFDIV	436	426	5.50%
DFMUL	376	366	6.38%
DFSIN	755	747	3.18%
MIPS	232	233	3.02%
ADPCM	541	546	1.66%
GSM	393	399	2.54%
JPEG	1,692	1,664	2.48%
MOTION	583	588	1.54%
AES	716	708	3.35%
BLOWFISH	1,406	1,412	0.71%
SHA	1,284	1,289	0.70%

are synthesizable even if modification according with the tool-specific extension is necessary. The definition also implies that except for the tool-specific extension, synthesizability of programs still depends on the HLS tool. This is because the synthesizable subset of the C language is different between tools. A mature state-of-the-art HLS tool can accept a large subset of the C language, while an immature tool can accept a very small subset.

We confirmed the synthesizability of the CHStone programs with a commercial HLS tool eXCite from YXI [18], which generates an RTL code in Verilog-HDL or VHDL from a C program. Although the vendor says that eXCite takes a standard C program as input, eXCite requires tool-specific extensions such as I/O declarations. Thus, we modified the CHStone programs so that the CHStone programs become acceptable by eXCite. **Table 3** describes the lines of code in the CHStone benchmark suite and the lines of code after the modification in the second and third columns, respectively. The rightmost column represents the ratio of the lines modified according with the tool-specific extension to the total lines of the CHStone programs. The modification of the CHStone programs was on average approximate 3.0%, all of which were for the tool-specific extension such as I/O declarations. It is then confirmed that all the CHStone programs are synthesizable by eXCite.

We also tested the synthesizability of C programs of the benchmark suites mentioned in Section 1, i.e., 1995 High Level Synthesis Design Repository (HLSynth95) [6], SPEC [9], EEMBC [10], and MediaBench [11]. HLSynth95 has in total 23 programs written in VHDL, Verilog-HDL, C, and so on, and eight out of them are C programs. We ran eXCite for the eight C programs and confirmed that six of them were synthesizable by eXCite [⋆1], while the other two were not synthesizable because they include the unsynthesizable data types and constructs by eXCite. For the C programs in SPEC, EEMBC, and MediaBench, instead of actually running eXCite, we examined these source programs against the users manual of eXCite. We confirmed that these C programs include the unsynthesizable data types and constructs by eXCite and are not synthesizable.

Since eXCite was the only HLS tool available to us, in this paper we confirmed the synthesizability of the CHStone programs by eXCite. It does not mean that eXCite is the only HLS tool by which the CHStone programs are synthesizable. CHStone excludes the unsynthesizable data types and constructs by most of the existing HLS tools so that the CHStone programs are synthesizable by other state-of-the-art HLS tools. To extensively evaluate the synthesizability of the CHStone programs, more HLS tools should be used. We expect CHStone users to test the synthesizability of the CHStone programs with various HLS tools in future.

4.2 Resource Utilization

This section analyzes the characteristics on resource utilization of the circuits synthesized from the CHStone benchmark programs with eXCite. In order not to depend on the specific clock frequency or target library, HLS was conducted in the following conditions; No resource constraint was specified; The constraint on the clock frequency was specified so that each functional unit completes in a single clock cycle; Operation chaining was disabled; No aggressive optimization was applied such as pipelining, loop unrolling, memory access optimizations, and so on. **Table 4** describes the number of states and the types and the numbers of hardware resources for each benchmark program. For ROM, SRAM,

⋆1 These synthesizable C programs consist of only 10-20 lines of code, which is considerably smaller than the CHStone programs.

© 2009 Information Processing Society of Japan

Table 4 The number of states and resource utilization in RTL description.

	DFADD	DFDIV	DFMUL	DFSIN	MIPS	ADPCM	GSM	JPEG	MOTION	AES	BLOWFISH	SHA
No. of states	35	52	25	121	19	155	236	815	306	451	351	136
32-bit Adders/Subtracters	4	4	4	4	3	16	18	14	11	10	9	10
Multipliers				1		4	1	4		1		
Dividers						1		1		1		1
Comparators ⋆1	9	8	6	13	33	9	4	10	5	5	6	5
Shifters					2	1	2	2	4			
64-bit Adders/Subtracters	5	3	3	6								
Multipliers		4	4	4	2 ⋆2							
Dividers		1		1								
Comparators	8	5	5	10								
Shifters	6	2	2	6								
ROMs (bits)	17,024	12,416	12,032	12,800	1,920	17,664	7,296	65,712	17,600	18,368	116,544	131,296
SRAMs (bits)					3,072	9,408	3,408	160,448	16,768	19,584	34,240	3,232
Registers (bits)	1,917	1,862	574	4,935	491	3,698	1,409	5,032	890	3,202	2,031	1,054

and registers, the total number of bits is depicted. The number of resources is small considering the size of the source code. This is because no parallelization technique was used such as pipelining and loop unrolling. If such optimization is applied, the designs require more resources and improve performance. Also, **Fig. 3** more clearly displays the resource utilization in each programs.

Table 4 depicts that the CHStone programs have different features on the synthesized circuits. First, the number of states varies from 19 to 815. This means that the CHStone programs have the different features in terms of the complexity of the synthesized circuits. Also, we observe that the CHStone programs are not small application programs as traditionally used in the past literature on HLS, but rather large applications.

Next, we see from Table 4 and Fig. 3 that the resource utilization widely varies. For example, BLOWFISH uses only two types of resources, while DFSIN uses eight types. The arithmetic programs use a lot of 64-bit resources, since as explained in the previous section these programs have many operations dealing with 64-bit integers. Finally, we observe that a program with the large number of states does not always use a large number of resources. For example, DFSIN uses the various types and the large numbers of resources compared with JPEG even though the number of states in DFSIN is smaller than that in JPEG. Note that the number of resources for each operation type does not depend on the size of

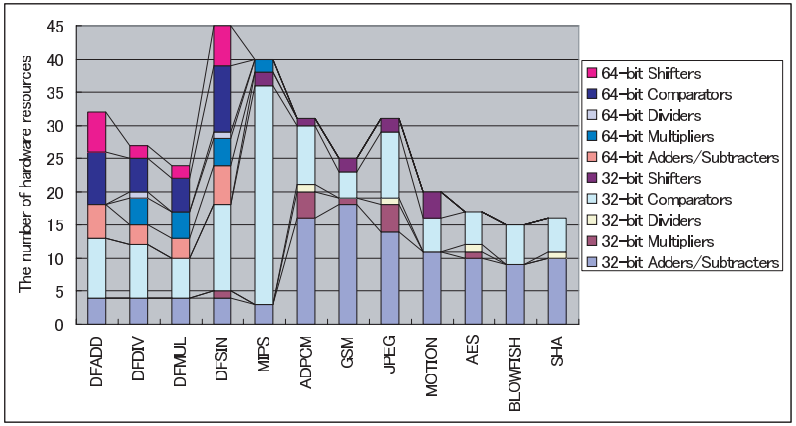

Fig. 3 Total resource utilization per benchmark program.

⋆1 In case of eXCite, 32-bit comparators are allocated not only to comparison operations shown in Table 2, but also to `case` statements.

⋆2 A MIPS multiplication instruction takes two 32-bit operands and generates a 64-bit result. eXCite allocates a 64-bit multiplier to the MIPS multiplication instruction.

© 2009 Information Processing Society of Japan

the input program, but depends on the operation-level parallelism. Also, ROM and SRAM in BLOWFISH are much larger than those in AES even though the number of states in BLOWFISH is smaller than that in AES.

The circuits synthesized from the CHStone programs have different characteristics in terms of resource utilization. This means that the benchmark programs in CHStone have the different characteristics at the source level.

4.3 Sensitivity to Resource Constraints

In general, the area and performance of the circuits generated by HLS are sensitive to resource constraints (i.e., constraints on the number of functional units) given to HLS tools. For example, by saving and sharing the resources, the area can be reduced. On the other hand, the number of states is increased, which leads to an increase in the execution cycles, i.e., performance degradation. In this section, we examine the area/performance sensitivity of the CHStone programs to resource constraints. Two resource constrains are given. One constraint is a maximum resource constraint, which aims to maximize the performance by giving sufficient resources. The other constraint is a minimum resource constraint, which aims to minimize the area by giving a single resource for each operation type. For logic synthesis and place-and-route, we used Synplify Pro from Synplicity [19] and XST from Xilinx [20], respectively. Xilinx Virtex 4 [20] was used as a target device.

Table 5 summarizes synthesis results under the two resource constraints. The number of states and hardware area such as the numbers of slices, built-in DSPs ⋆1, block-RAMs, and registers are described in Table 5. The hardware area will be discussed in the next section. In the second column, *min* and *max* ⋆2 represent the minimum resources and the maximum resources, respectively. We cannot disclose the number of execution cycles due to the license agreement of the synthesis tool. Instead, **Fig. 4** depicts the relative area and the number of execution cycles under the maximum resource constraint against those under the minimum resource constraint ⋆3.

As mentioned above, in general the performance of the circuits generated by HLS is sensitive to resource constraint. Actually, Fig. 4 shows that the performance of the CHStone programs is also sensitive to resource constraints. Furthermore, Fig. 4 shows that the CHStone programs have the different sensitivity to resource constraints. In general, if a program has the high operation-level parallelism, the generated circuit uses a large number of resources sufficient to minimize the number of execution cycles under a maximum resource constraint. On the other hand, when synthesizing from the same program under the minimum resource constraint, the number of execution cycles is increased. Figure 4

Table 5 Synthesis results under different resource constraints.

	Resource const.	States	Slices	DSPs	Block-RAMs		Registers (bits)
					16 × 1	32 × 1	
DFADD	min	45	5,056				1,857
	max	35	5,903				1,986
DFDIV	min	64	4,966	4			1,915
	max	52	5,006	16			1,960
DFMUL	min	31	2,292	4			730
	max	25	2,240	16			599
DFSIN	min	153	18,443	7			5,441
	max	121	18,269	19			5,016
MIPS	min	47	1,588	14		192	911
	max	19	1,412	14		192	563
ADPCM	min	227	8,445	3	320	64	3,866
	max	155	8,066	12	320	64	3,776
GSM	min	268	4,166	1	96		1,646
	max	236	5,182	1	96		1,512
JPEG	min	950	15,372	3	304	320	5,105
	max	815	16,366	10	304	320	5,056
MOTION	min	341	5,258		64		999
	max	306	3,330		64		919
AES	min	475	9,034	3	32	256	3,539
	max	451	9,287	3	32	256	3,624
BLOWFISH	min	357	5,629		96	96	2,369
	max	351	4,604		96	96	2,116
SHA	min	138	5,916		64		1,113
	max	136	6,108		64		1,058

⋆1 DSPs are used to implement multipliers.

⋆2 Registers in *max* rows in Table 5 are larger than those in Table 4 since logic synthesis and place-and-route tools duplicate registers to improve the performance.

⋆3 The hardware area includes slices and DSPs. The area of DSPs is converted to an equivalent slice count.

© 2009 Information Processing Society of Japan

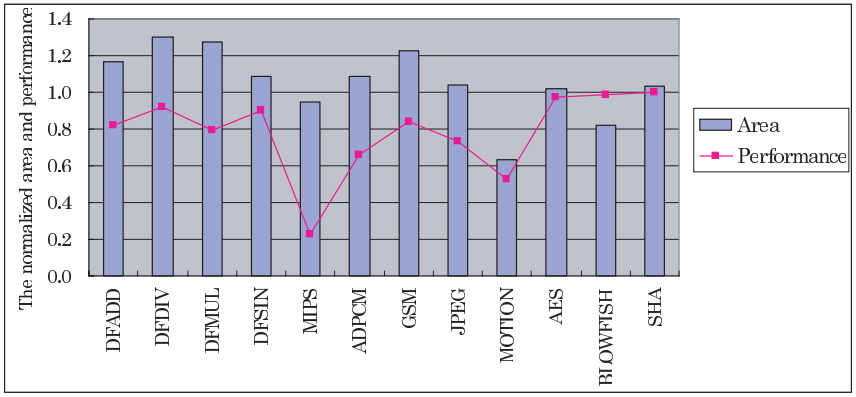

Fig. 4 Sensitivity of area and execution cycles to resource constraint.

shows that in AES, BLOWFISH, and SHA the numbers of execution cycles under the maximum resource constraint are hardly decreased compared with those under the minimum resource constraint. This is because these programs have the low operation-level parallelism and are not very sensitive to resource constraints. On the other hand, MIPS is very sensitive to resource constraints. Figure 4 shows that in MIPS the number of execution cycles under the maximum resource constraint is small by approximate 75% compared with that under the minimum resource constraint. Also, MOTION is sensitive to resource constraints and has an interesting feature as follows. In MOTION, when comparing the synthesis results under the maximum resource constraint with those under the minimum resource constraint, the number of execution cycles in Fig. 4 is reduced by approximately 50% in spite of only 10% difference in the numbers of states in Table 5. This is because the operation-level parallelism of a few kernel loops in MOTION is rich.

We have seen above that the CHStone benchmark programs have different characteristics in sensitivity to resource constrains as well as resource utilization, as shown in Section 4.2.

4.4 Challenges for Practical High-Level Synthesis

Our experimental analysis in Section 4.3 implies a few challenges to be solved in the future in order for HLS to be a really practical technology.

Table 5 and Fig. 4 indicate that resource sharing by giving tight resource constraints does not always reduce the overall area. Indeed, resource sharing reduces the number of functional units. However, the overall area is increased in some cases. This fact is mainly due to two reasons as follows. One reason is additional multiplexers. In order to share hardware resources, multiplexers often need to be inserted at the input ports of the shared resources. The other one is the increased control-path area. As shown in Table 5, the tight resource constraint leads to an increase in the number of states in the controller, and in general the control-path area heavily depends on the number of states. The area of multiplexers and control-path is negligible when input behavioral descriptions are small, and in actual fact, most of the research efforts on HLS in the 1980s and the early 1990s aimed at the minimization of the number of resources. If we synthesize hardware from large behavioral descriptions, however, the area of multiplexers and control-path is not negligible, and inexpensive resources should not be shared aggressively. Future HLS technology should find the optimal balance between resource sharing and unsharing in order to reduce the overall hardware cost.

Through our experiments using CHStone, we found that the overall critical path delay of the HLS-generated circuits is often much longer than the functional unit delay on the critical path of the datapath. We cannot disclose the actual critical path delay because of the license agreement of the synthesis tools. Instead, **Fig. 5** shows the relative critical path delay against the functional unit delay on the longest datapath. The critical path delay in the figure is based on static timing analysis after logic synthesis and place-and-route. The logic synthesis and place-and-route are performed with the goal of critical path delay minimization. Figure 5 shows that the overall critical path delay is up to 5.3 times longer than the functional unit delay[⋆1]. This means that delays of other components such as multiplexers, control-path and interconnections are significant[⋆2]. In our experiments presented in this paper, we could not analyze the delays of individual

⋆1 For logic synthesis, we used commercial IPs of functional units written in HDL. In case of SHA, comparators had the longest functional unit delay of all the hardware components.

⋆2 One may think that the difference between the critical path delay and the functional unit delay in Fig. 5 changes depending on the constraint on the clock frequency given to the HLS tool. In Fig. 5, however, we specified some constraints to the HLS tool so that generated circuits do not depend on the constraint on the clock frequency.

© 2009 Information Processing Society of Japan

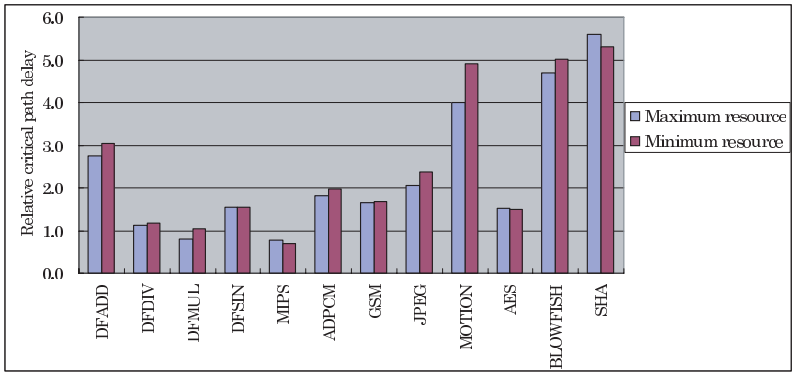

Fig. 5 Relative critical path delay against the longest functional unit delay.

components in detail since we flattened all the components (except DSPs) into the gate level and then optimized the gate-level circuit together with retiming enabled ⋆1. In future, we should analyze the critical path delay in detail. Still, it should be noted that although a reasonable amount of prior works on HLS exist, which consider the interconnect and multiplexer delays, few works focus on the controller delay. Since the controller delay cannot be ignored when input behavioral descriptions are large [8),21)], this should be taken into account together with delays of functional units, multiplexers, and interconnections in the future.

Although the two problems mentioned above are known problems, they have not been taken into account sufficiently in the past. Indeed, the problems used to be trivial when the input programs were small. The experimental results in this section quantitatively reveal the importance of the two problems by using the CHStone programs. CHStone will contribute to the further advance of the HLS technology so that HLS can be a really practical solution for designing complex SoCs in a short time.

⋆1 This is why the overall critical path delay is shorter than the functional unit delay in a few cases.

5. Discussion

This section discusses the novelty and usefulness of CHStone.

At present, it is difficult to objectively evaluate the novelty and usefulness of CHStone. This is due to two reasons as follows. One reason is that there are no C-based HLS benchmark suites with which CHStone should be compared. The other is that the HLS community has not established a common recognition yet on what are sufficient and necessary requirements for C-based HLS benchmark programs to meet. Instead, when developing CHStone, we emphasized (1) diversity, (2) size, (3) synthesizability, and (4) usability since we believe that these four features are significantly important for C-based HLS benchmark programs.

On the four points above, in Sections 2, 3, and 4, we have analyzed CHStone from the several aspects and shown that (1) the CHStone programs have the various characteristics in terms of the application domains, the source-level characteristics such as the types of operations and control structures, the required resources in generated RTL circuits, and the sensitivity to resource constraints, (2) CHStone has practically large programs as judged from the size of source-level descriptions (the lines of code, the numbers of functions, variables, and operations) and generated RTL circuits (the number of states, the types and numbers of functional units, and the size of memories in generated circuits), (3) the CHStone programs are synthesizable by a commercial HLS tool and the modification of the programs was on average approximate 3.0% out of the total lines of code, and (4) CHStone users can easily and quickly use the CHStone programs since test vectors are self-contained and no external library is necessary, and CHStone users can freely change the descriptions of the CHStone programs since the source code of the CHStone programs has been released to the public.

As stated above, this paper has shown quantitative analysis results of CHStone from the several aspects. Here, the following questions may arise:

(i) Does CHStone sufficiently meet the requirements on the aforementioned four points?
(ii) Are the analysis results reliable?
(iii) Are there any other points to be taken into account for developing benchmark programs for C-based HLS?

© 2009 Information Processing Society of Japan

As mentioned above, in this paper, we have analyzed the CHStone programs on diversity, size, synthesizability, and usability. It is still unclear if the CHStone programs sufficiently meet the requirements on these four points. For example, it might be better to include more benchmark programs in CHStone in order to cover the wider diversity. The number of programs in CHStone is small compared with benchmark suites in other fields. For example, the SPEC and EEMBC suites, which are developed to evaluate general-purpose and embedded processors, are composed of dozens of benchmark programs. Note that, in our review of the recent literature on HLS, few works used more than 12 benchmark programs which are as large as the ones in CHStone. Thus, considering the current status of the world-wide researches on HLS, the CHStone programs are practically large and diverse.

To the second question, it is reasonable to say that the analysis results shown in this paper are reliable. This is because eXCite, which is a commercial HLS tool that we used in this paper, is a proven HLS tool which has been widely used not only in academia but also in industry ⋆1. The analysis results might vary depending on the tool but the difference is not so large from our experience.

The last question is whether there are other points to be taken into account when developing benchmark programs for C-based HLS. For example, some readers, especially LSI designers in industry, would like human-designed RTL circuits and/or their data to be released with benchmark programs for C-based HLS so that the synthesized circuits through HLS can be compared with the RTL circuits. However, this is not always a necessary requirement for C-based HLS benchmark programs to meet. CHStone users are mainly expected to be HLS researchers. Most HLS researchers do not always need to compare the synthesized circuits through HLS with human-designed RTL circuits. Instead, they are interested in using benchmark programs for the evaluation of the effectiveness of their proposed techniques, e.g., the comparison of results with their techniques and those without their techniques. This is obviously shown in the fact that very few works on HLS compare results of the synthesized circuits through HLS and those of the human-designed RTL circuits in their papers. Also, there might remain some other points to be considered. As previously mentioned, however, at present it is difficult to objectively discuss this matter since there is no common recognition on the sufficient and necessary requirements for C-based HLS benchmark programs to meet. We expect that CHStone will be a first step for the HLS community to establish such common recognition for C-based HLS benchmark programs.

To the best of our knowledge, CHStone is the first benchmark suite which consists of real-world applications, is large in size, presents various characteristics in the source and RT levels, is refined in an HLS-friendly manner, and is available to the public. Also, all of these features make the CHStone benchmark suite highly useful. Additionally, the source- and RT-level analysis results presented in this paper will be useful for HLS researchers to efficiently analyze their experimental results using the CHStone programs.

6. Summary and Current Status

This paper proposed CHStone, a suite of benchmark programs for C-based high-level synthesis. CHStone consists of a dozen of large and easy-to-use programs written in C, which are selected from various application domains. We expect that CHStone will be widely used by HLS researchers and can contribute to the further advance of the HLS technology. This paper also analyzes the CHStone benchmark programs in terms of the source-level characteristics, the resource utilization, the sensitivity to resource constraints, and so on. These analysis results will be useful for HLS researchers to analyze the effectiveness of their new techniques in experimental results using the CHStone programs. In addition, we revealed future challenges towards the practical high-level synthesis. CHStone has been already released for free at Ref. 12).

Acknowledgments This work is in part supported by KAKENHI 19700040.

⋆1 It is preferable to use multiple tools for more objective analysis. At present, unfortunately, no other HLS tool is available to us. Even if available, however, it would be difficult for this paper to provide tangible data analyzed by multiple tools due to the license agreement of each tool.

References

1) Gajski, D.D., Dutt, D.N., Wu, C.-H.A. and Lin, Y.-L.S.: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

© 2009 Information Processing Society of Japan

2) Wakabayashi, K. and Okamoto, T.: C-based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.19, No.12, pp.1507–1522 (2000).
3) Gupta, S., Gupta, K.R., Dutt, D.N. and Nicolau, A.: *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers (2005).
4) De Micheli, G.: Hardware Synthesis from C/C++ Models, *Proc. Design, Automation and Test in Europe*, IEEE Computer Society, pp.382–383 (1999).
5) Dutt, D.N. and Ramchandran, C.: Benchmarks for the 1992 High Level Synthesis Workshop, Technical Report 92–107, University of California, Irvine (1992).
6) Panda, R.P. and Dutt, D.N.: 1995 High Level Synthesis Design Repository, *Proc. International Symposium on System Synthesis*, IEEE Computer Society, pp.170–174 (1995).
7) Vahid, F.: Partitioning Sequential Programs for CAD Using a Three-Step Approach, *ACM Trans. Design Automation of Electronic Systems*, Vol.7, No.3, pp.413–429 (2002).
8) Hara, Y., Tomiyama, H., Honda, S., Takada, H. and Ishii, K.: Function-Level Partitioning of Sequential Programs for Efficient Behavioral Synthesis, *IEICE Trans. Fundamentals*, Vol.E90-A, No.12, pp.2853–2862 (2007).
9) Standard Performance Evaluation Corporation: SPEC – Standard Performance Evaluation Corporation (online). http://www.spec.org/ (accessed 2009-07-07).
10) The Embedded Microprocessor Benchmark Consortium: EEMBC – The Embedded Microprocessor Benchmark Consortium (online). http://www.eembc.org/ (accessed 2009-07-07).
11) Lee, C., Potkonjak, M. and Mangione-Smith, H.W.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *Proc. International Symposium on Microarchitecture*, IEEE Computer Society, pp.330–335 (1997).
12) CHStone: CHStone Home Page (online). http://www.ertl.jp/chstone/ (accessed 2009-07-07).
13) Hauser, J.: SoftFloat (online). http://www.jhauser.us/arithmetic/SoftFloat.html (accessed 2009-07-07).
14) SNU Real-Time Benchmarks: SNU Real-Time Benchmarks (online). http://archi.snu.ac.kr/realtime/benchmark/ (accessed 2009-07-07).
15) Hung, C.A.: PVRG-JPEG CODEC 1.1, Technical Report, Stanford University (1993).
16) AILab: A.I.Lab Web Site (online). http://www-ailab.elcom.nitech.ac.jp/ (accessed 2009-07-07).
17) Guthaus, R.M., Ringenberg, S.J. and Ernst, D.: MiBench: A free, Commercially Representative Embedded Benchmark Suite, *Proc. Workshop on Workload Characterization*, IEEE Computer Society, pp.3–14 (2001).
18) Y Explorations, Inc.: YXI - C to RTL Behavioral Synthesis (online). http://www.yxi.com/ (accessed 2009-07-07).
19) Synplicity: Synplicity: Home (online). http://www.synplicity.com/ (accessed 2009-07-07).
20) Xilinx Inc.: FPGA and CPLD Solutions from Xilinx Inc. (online). http://www.xilinx.com/ (accessed 2009-07-07).
21) Gupta, R.G., Gupta, M. and Panda, R.P.: Rapid Estimation of Control Delay from High-Level Specifications, *Proc. Design Automation Conference*, ACM/EDAC/IEEE, pp.455–458 (2006).

(Received March 2, 2009)
(Accepted July 2, 2009)
(Released October 7, 2009)

Yuko Hara received her B.E. in Information Engineering and her M.E. in Graduate School of Information Science from Nagoya University in 2006 and 2008, respectively. Currently she is a Ph.D. candidate at Graduate School of Information Science, Nagoya University. Her research interests include behavioral synthesis and embedded systems. She is a member of ACM and IPSJ.

© 2009 Information Processing Society of Japan

Hiroyuki Tomiyama received his Ph.D. degree in computer science from Kyushu University in 1999. From 1999 to 2001, he was a visiting postdoctoral researcher with the Center of Embedded Computer Systems, University of California, Irvine. From 2001 to 2003, he was a researcher at the Institute of Systems & Information Technologies/KYUSHU. In 2003, he joined the Graduate School of Information Science, Nagoya University, as an assistant professor, where he is now an associate professor. His research interests include design automation, architectures and compilers for embedded systems and systems-on-chip. He currently serves as an editorial board member of IPSJ Transactions on SLDM, IEEE Embedded Systems Letters, and International Journal on Embedded Systems. He has also served on the organizing and program committees of several premier conferences including ICCAD, ASP-DAC, DATE, CODES+ISSS, and so on. He is a member of ACM, IEEE, IPSJ and IEICE.

Shinya Honda received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005. From 2004 to 2006, he was a researcher at the Nagoya University Extension Course for Embedded Software Specialists. In 2006, he joined the Center for Embedded Computing Systems, Nagoya University, as an assistant professor. His research interests include system-level design automation and real-time operating systems. He received the Best Paper Award from IPSJ in 2003. He is a member of IPSJ.

Hiroaki Takada is a Professor at the Department of Information Engineering, the Graduate School of Information Science, Nagoya University. He is also the Executive Director of the Center for Embedded Computing Systems (NCES). He received his Ph.D. degree in Information Science from the University of Tokyo in 1996. He was a research associate at the University of Tokyo from 1989 to 1997, and was a lecturer and then an associate professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IPSJ, IEICE, and JSSST.

© 2009 Information Processing Society of Japan