GCC2Verilog Compiler Toolset for Complete Translation of C Programming Language into Verilog HDL

Giang Nguyen Thi Huong and Seon Wook Kim

Reconfigurable computing using a field-programmable gate-array (FPGA) device has become a promising solution in system design because of its power efficiency and design flexibility. To bring the benefit of FPGA to many application programmers, there has been intensive research about automatic translation from high-level programming languages (HLL) such as C and C++ into hardware. However, the large gap of syntaxes and semantics between hardware and software programming makes the translation challenging. In this paper, we introduce a new approach for the translation by using the widely used GCC compiler. By simply adding a hardware description language (HDL) backend to the existing state-of-the-art compiler, we could minimize an effort to implement the translator while supporting full features of HLL in the HLL-to-HDL translation and providing high performance. Our translator, called GCC2Verilog, was implemented as the GCC's cross compiler targeting at FPGAs instead of microprocessor architectures. Our experiment shows that we could achieve a speedup of up to 34 times and 17 times on average with 4-port memory over PICO microprocessor execution in selected EEMBC benchmarks.

Keywords: HLL-to-HDL translator, GCC2Verilog, FPGA, compiler.

Manuscript received Nov. 2, 2010; revised Aug. 11, 2011; accepted Jan. 27, 2011.

This work was supported by Seoul R&BD Program (10920) and the Basic Science Research Program through National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0010262).

Giang Nguyen Thi Huong (phone: +82 2 3290 3794, email: redriver@korea.ac.kr) and Seon Wook Kim (corresponding author, email: seon@korea.ac.kr) are with the Department of Electrical Engineering, Korea University, Seoul, Rep. of Korea.

http://dx.doi.org/10.4218/etrij.11.0110.0654

I. Introduction

Many system developers have convincing reasons to add a field-programmable gate array (FPGA) to their design platform, and often replace traditional general-purpose processors or digital signal processors (DSPs) with FPGAs due to significantly better computational performance per watt over microprocessors and design flexibility over ASICs [1]-[4]. However, the task of configuring FPGAs with a description of hardware description language (HDL) like Verilog, VHDL, and SystemC requires substantial amount of knowledge in hardware design methods, which makes the potential advantages of the FPGA computing unrealizable to most software developers. Therefore, there is an increasing demand to design hardware at a higher abstraction level such as software programming languages without being concerned about hardware-specific details. This approach requires a translation tool that generates the HDL codes from commonly used high-level programming languages (HLL) in software development [5]-[8]. However, the large gap between hardware (HW) and software (SW) programming concepts makes the translation challenging [9]. For example, the view of memory as simple bit vectors and the absence of hierarchical control flows in function calls on hardware side make it difficult to translate software's complex data types, such as multidimensional arrays, structures, or pointers, and a deeply nested function calls into HDL automatically.

There has been intensive research on the HLL, especially C or C-like language, to HDL translation; however, none of the work supports all the ANSI C syntaxes [10]. Therefore, a lot of reprogramming efforts are needed to make existing C codes compliant with the translators [6], [7], [11]-[13]. However, the efforts were not always successful. In [10], the authors tried to

rewrite 158 testing functions to make them compatible with several state-of-the-art translators, but they only succeeded in 45 cases. Moreover, the construction of the HLL-to-HDL translator was not a simple task. Some researchers built their own compiler systems from scratch [5], [8]. Others employed a SW compiler front-end, such as Stanford University Intermediate Format (SUIF) [14], to parse HLL codes and build their own intermediate representation (IR) for code optimization and HDL code generation [7], [15]. We realized that the incomplete translation comes from the fact that the translators have been built from a hardware designer's view.

In this paper, we show that we can build the translator, called GCC2Verilog, based on a widely used general HLL compiler such as GCC [16], while supporting all C syntaxes in the translation and providing high performance in the translated HW codes at the same time. For our compiler, we modified the GCC's final code generation pass and changed a target machine description to aim at the FPGA instead of microprocessors. The GCC compiler offers a good infrastructure for the implementation of an HLL-to-HDL translator. Most SW compiler's optimizations are beneficial for the HDL translation; the low-level IR of an HLL compiler resembles with HDL languages and can be easily translated into HDL codes. Therefore, by using the existing open source compiler, we could minimize the effort to implement our translator and support all the ANSI C syntaxes in the translation without any rewritten requirement. The GCC2Verilog directly translates the final version of GCC's IR, that is, register transfer language (RTL), into synthesizable Verilog code, right before the RTL code is translated into assembly code. Additionally, to support unlimited nesting cross calls between software and hardware and even recursive calls in hardware, the GCC2Verilog compiler generates the hardware codes to have a stack and to follow a host processor's linkage convention. For our execution framework, we used the processor in interactive play with compiler (PICO) [17] as the host processor which is an in-house one-way in-order embedded microprocessor and has comparable performance with ARM9. The interface of PICO was optimized to enable very low communication overhead with HW intellectual properties (HWIPs) in FPGA, which are automatically generated from C by our compiler system. Our experiment shows that we could achieve a speedup of up to 34 times and 17 times on average with 4-port memory over the PICO microprocessor execution in selected EEMBC benchmarks.

Our paper provides three main contributions: i) proof that an existing HLL compiler system such as GCC is a good infrastructure to build an HLL-to-HDL translator with very insignificant implementation effort; ii) a presentation of a detailed implementation of translation from GCC's RTL into Verilog language and the method to support all kinds of C language features; and iii) a demonstration that such a compiler system can achieve good performance in the translated HW codes.

The rest of this paper is organized as follows. Our related work is discussed in section II. Section III explains our compilation and execution environment. Section IV presents the implementation of the GCC2Verilog compiler. Section V describes the Verilog code generation in detail. We evaluate the performance of our compiler system in section VI. Finally, we make conclusion in section VII.

II. Related Work

Many researches employing different approaches have worked on the HLL-to-HDL translation to overcome the gap between hardware and software programming concepts. Some research, instead of supporting the commonly used HLL, used an approach to define new HLL-like languages which contain a subset of common HLL constructs with some extension to control hardware instantiation and to support parallelism. For example, Impulse-C [5], Handel-C [8], and Transmogrifier C [18] belong to this category. However, many common HLL programming concepts, such as pointers, structures, recursive function calls, and irregular control statements, are not allowed in these languages. Also, the languages are unfamiliar to most of HLL programmers because of the requirement of hardware design understanding.

Other researchers have tried to use only a subset of particular high-level languages (normally C/C++) for HDL translation [6], [7], [11]-[13], [19]. Mentor Graphics' Catapult C [19] performs behavioral synthesis from a strict ANSI C++ subset, that is, C++ programs should be written in a synthesizable C++ style. Pointers must be statically determined, and therefore memory (de)allocation is not supported. Similarly, the Spark project [6] implemented the C to VHDL translator without supporting common C constructs like pointers, 2D arrays, or irregular control statements. However, even with the supporting code types completely rewriting source codes is necessary to make them efficiently translated [7], [20]. An empirical comparison on automation capacity of several common HLL-to-HDL translators, such as Spark [6], ROCCC [7], and DWARV [13], was presented in [10]. The translators were evaluated qualitatively in terms of the supported ANSI C subsets, restrictions required on the subsets, the rewritten effort needed to make existing C codes compliant with each translator, the requirement of HW knowledge, and the tools' testability and readability. The results showed that these translators supported only 51% to 63% of the C language features; the restricted syntaxes in input codes include a lot of commonly used data

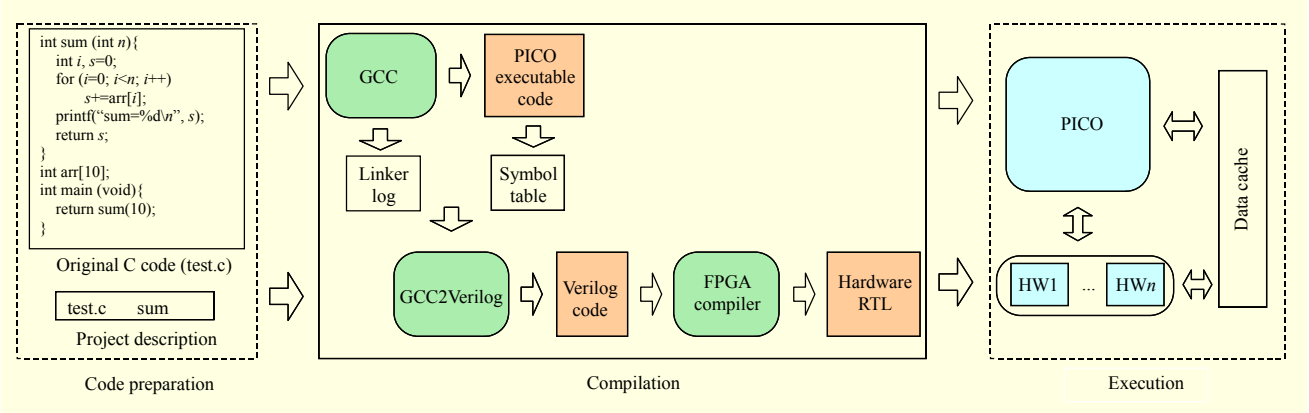

Fig. 1. GCC2Verilog's compilation flow and execution framework.

types (float, structure, union), operators (field selection, object address operator), control statements (while-loop, break, continue, goto), and even function calls [10]. Also, there were many additional restrictions in use of the supported syntaxes. For example, the ROCCC compiler requires that for-loops must be perfectly nested, the indices of arrays used inside a loop must be equal to a loop counter, and the number of iterations must be equal to the number of elements of an output array. These restrictions make the translators far from convenient tools for software programmers.

Overall, pointers and function calls are the most difficult syntaxes to be translated into hardware. In [21], in order to support pointers, the authors first declared a universal memory, and partitioned the memory which each variable refers by applying pointer analysis. The pointers were then synthesized by encoding their values and generating circuits to dynamically access locations they may reference. However, a pointer cannot contain the address of a dynamically allocated memory whose size is unknown at compile time. For function call, the HybridThreads [22] and ASH [15] allowed function calls inside hardware code, but it could not support a cross call from hardware back to software.

III. Compilation and Execution Framework

The overview of our compilation and execution framework is shown in Fig. 1, which consists of three phases: code preparation, compilation, and execution. Our framework translates C code at procedure granularity into Verilog modules, that is, function-by-function. For the translation, a programmer needs to identify a function name and its containing source file name in a project description file at the code preparation phase. An example of the project description file is shown in Fig. 1, in which sum function in test.c source file will be translated into Verilog code. The subroutine form in C allows each HWIP to have its own stack space, which supports seamless cross calls between software and hardware. Also, if there is constraint on the number of available registers, we can spill their values onto the stack like an HLL compiler.

The compilation phase consists of three steps: software compilation, C-to-Verilog translation, and hardware synthesizing.

- Software compilation: The GCC compiler of a host processor (the GCC PICO compiler) compiles an entire input source code, that is, software and hardware codes together, for the software execution in order to share addresses of data and instructions with hardware, that is, automatic address resolution in software with hardware. Even though hardware functions are not executed in a microprocessor, they should be included in the software executable code. Otherwise, some software functions accessed by the hardware may be excluded from the executable code by GCC's optimization. For example, the printf library function in Fig. 1 is used only from the hardware sum function but not from any software function. The printf will be excluded from the executable code by a software linker. Therefore, when hardware calls printf for its software execution, the target function cannot be found.
- C-to-Verilog translation: Our GCC2Verilog compiler translates the registered functions in the project description file into synthesizable Verilog codes. The hardware compilation needs two inputs, a symbol table and a linking log file from the software compilation in order to perform software address resolution, which will be explained in the next section. The software codes and the generated hardware codes follow the same linkage convention for efficient cross calls between them.
- Hardware synthesizing: The generated Verilog code is synthesized by the FPGA compiler into hardware bitstreams; hardware functions become configured hardware IPs on FPGA while the executable code including software functions runs on the PICO processor.

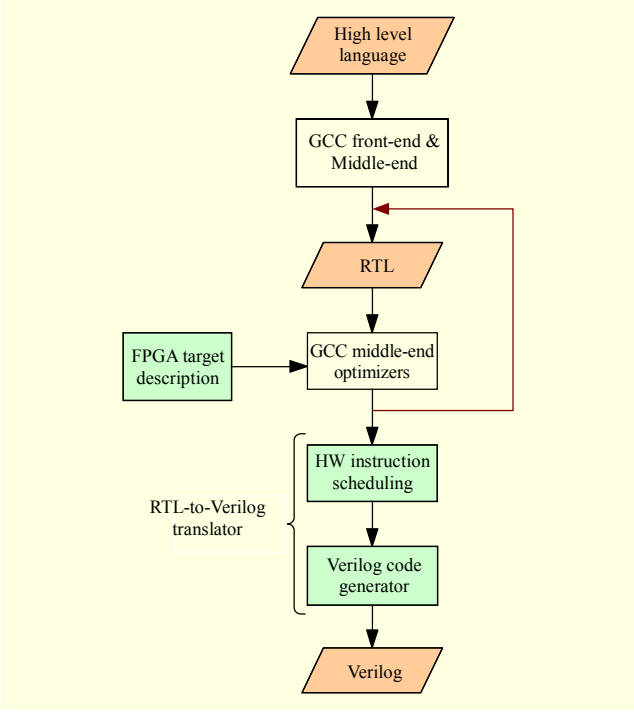

Fig. 2. GCC2Verilog infrastructure with GCC's front-end, middle-end, and FPGA backend with additional components (colored ones) for Verilog code generation.

At the execution phase, software and hardware codes communicate with each other through function calls, and they use the same memory hierarchy since they use one address space. In the current implementation, the PICO processor and HWIPs are not concurrently executed; therefore, we do not need to consider the memory coherence problem.

IV. GCC2Verilog Implementation

1. Configuring FPGA Backend

The GCC2Verilog was implemented as a GCC's cross compiler by adding a new FPGA target machine and an RTL-to-Verilog code generator. Figure 2 shows the additional components of GCC2Verilog attached to the conventional GCC infrastructure to perform the Verilog code translation. The configuration of FPGA target was used during the RTL code optimization passes and the code generation. After performing all optimizations specific to the FPGA backend, the final RTL code was passed to the RTL-to-Verilog translator which consists of the HW instruction scheduler and the Verilog code generator.

GCC is a multitarget compiler, and therefore the work of porting a new target backend can be done very easily. The FPGA targeted machine was assumed as a twenty-way in-order microprocessor with plenty of registers. The number of execution ways and registers are redundantly defined so that the compiler can aggressively optimize the code for the best performance, that is, exploiting higher instruction-level parallelism. The number of functional units and registers actually used in an HWIP generated from the GCC2Verilog compiler is dependent on the application's characteristics. The number of used registers is minimized by register allocation in the GCC pass. Also, we have a lot of flexibilities in code generation when comparing a traditional microprocessor backend since there is no constraint in designing the instruction set architecture (ISA) of the FPGA target (or Verilog backend). For example, in order to reduce intermediate operations, that is, reduce the number of instructions, we tried to combine add/sub, logical, and shifts operations since these operators can be executed together in one clock cycle due to their short execution latency. The following instructions are some examples: $(a + b) >> d$, $a + b - c$, $(a \,\&\, b) << d$, and $a - (b \,\hat{}\, d)$.

2. Utilizing the GCC's RTL for Verilog Code Translation

The GCC's RTL intermediate representation (IR) can be easily used for HDL code generation. An RTL instruction is constructed with a 3-address form to describe a functional unit and its operands (stored in registers or memory) for execution on a target machine [23], and therefore, the IR can be easily mapped to Verilog expression. The GCC2Verilog utilizes the GCC's last RTL version, right before it is translated into assembler code, as input of the Verilog code translation. Therefore, we can take advantage of rich optimization techniques in GCC such as deadcode elimination, constant propagation, and loop unrolling. In GCC's RTL IR, all accesses to C complex data types such as pointers and structures are converted into simple memory accesses; all complex control statements are lowered to jump instructions. So, the burdensome problems of supporting pointers, irregular control statement, and complex data types of other HLL to HDL translation researches are naturally resolved.

We applied the finite state machine (FSM) model to the generated Verilog code so that it preserves the software execution order while exploiting concurrency in hardware execution. A function's RTL sequences are divided into several FSM states by considering control dependence, data dependence, and memory constraint, that is, the number of allowed concurrent memory accesses, and they are scheduled at compilation time, that is, statically by the HW instruction scheduler in Fig. 2. A state of the FSM represents an execution of one clock cycle. Every arithmetic instruction also takes only one cycle in the hardware, and independent instructions are wrapped within one FSM state for parallelism exploitation. Some instructions such as function calls and

Table 1. Examples of PICO assembler code and corresponding generated Verilog code of GCC2Verilog translator.

Operations	PICO assembler	Verilog code	
		Datapath	Control unit
Addition	add3 $r0, $r2, $r8	reg0 <= reg2 + reg8;	
Multiplication	mult3 $r0, $r2, $r8	reg0 <= reg2 * reg8;	
Shift left	asli $r3, #1	reg3 <= reg3 <<< 1;	

Memory load	ld.w $r2, $r5	always@(posedge clk or negedge reset)
		if(!mem_stall) begin
		read <= 0;
		case (pc)
		0: begin
		addr <= reg5;
		read <= 1'b1; be <= 4'b1111;
		end
		1: reg4 <= rdata;
		…
		endcase
		end
			always@(*)
			if(!mem_stall)
			case (pc)
			0: nx_pc <= 1;
			1: nx_pc <= 2;
			…
			endcase
			always@(posedge clk or negedge reset)
			begin
			….
			pc <= mem_stall ? pc : nx_pc;
			end

Conditional branch	cmpi $r2, 9
	blt.L7
			assign successor = (reg2 <= 9);
			always@(*)
			if(!mem_stall)
			case(pc)…
			x: nx_pc <= successor ? dest1 : dest0;
			endcase

Table 2. Example of code generation using one and two memory ports of FPGA backend.

Assembly code	Generated Verilog code	
	One-port memory	Dual-port memory

1. rsh $r7, #2
2. st.w ($r8+4), $r7
3. ld.w $r4, ($r0)
4. add3 $r0, $r8, $r0
5. mult $r7, $r3, $r3
	case (pc)
	0: begin
	reg7 <= reg7 >> 2;
	addr <= reg8 + 4;
	write <= 1'b1;
	be <= 4'b1111;
	pc <= 1;
	end
	1: begin
	wdata <= reg7;
	addr <= reg0;
	read <= 1'b1;
	be <= 4'b1111;
	reg0 <= reg8 + reg0;
	reg7 <= reg3 * reg3;
	pc <= 2;
	end
	2: reg4 <= rdata;
	…
	endcase
		case (pc)
		0: begin
		reg7 <= reg7 >> 2;
		addr0 <= reg8 + 4;
		write0 <= 1'b1;
		be0 <= 4'b1111;
		addr1 <= reg0;
		read1 <= 1'b1;
		be1 <= 4'b1111;
		reg0 <= reg8 + reg0;
		pc <= 1;
		end
		1: begin
		wdata0 <= reg7;
		reg7 <= reg3 * reg3;
		reg4 <= rdata1;
		pc <= 2;
		end
		…
		endcase

memory operations take several cycles; they are executed in several continuous FSM states. Table 1 compares some PICO's assembler instructions with the corresponding generated Verilog codes. The Verilog codes for common arithmetic and logical operations are similar to the PICO assembly codes. The load instruction is executed in two FSM states: the first state issues a memory service with an address (addr) and control signals (be, read), and the second state checks the readiness of the read data and writes the data into a register (reg4 <= rdata).

Table 2 shows the details of memory instruction scheduling. Instructions 2 and 3 are independent, but they are serialized if one-port memory is used. The hardware instruction scheduler divides one memory RTL into two FSM states and reschedules them for exploiting higher instruction level parallelism (ILP). Instruction 2 is truly dependent on instruction 1; however, instruction 2's Verilog code for memory address preparation and control handling are independent on instruction 1; therefore, these Verilog statements can be executed together with instruction 1 at the same state.

Our compiler can generate the Verilog code from any C code. Therefore, we can also translate codes to contain dynamic pointers or memory management functions like *malloc* into Verilog without any limitation. However, we did not try to translate the functions into hardware due to execution in efficiency since the memory management is better performed by OS. The memory allocation is done by SW, that is, OS, and then the allocated memory address is shared by HW through

the HW and SW calling method in [24]. The memory deallocation is done in the similar way. We also did not implement floating-point functional units inside FPGA due to resource issue; whenever a floating-point operation needs to be performed, the GCC2Verilog calls a library function of GCC (libgcc) in software. The non-blocking assignment (<=) is used in the generated Verilog code to avoid any kind of race condition. In this paper, we focus only on the C to Verilog translation, but with only minor changes, our compiler system is able to support translation of other HLLs supported by GCC, such as FORTRAN, C++, and Java, into Verilog.

3. Address Resolution between Hardware and Software

Since only one address space is used for both software codes and hardware codes, addresses of global variables as well as target/return addresses of software and hardware functions should be shared between their codes. In order to perform a call, a caller must know a callee's address in software or hardware. Also, the callee should know the return address of the caller for correctly restoring the caller's execution. However, in case of a cross call (SW calls HW or HW calls SW), the hardware does not know addresses in software and vice versa because they are separately compiled. Therefore, we need to resolve the software addresses for hardware execution and hardware addresses for software execution.

The hardware address resolution is done by assigning a unique hardware identification number (HWID) to each hardware function at the compilation phase; this work is automatically done by the PICO and GCC2Verilog compilers. The HWID is considered as an access point to a hardware function, since the hardware does not use addresses like in the software. A software function will use the HWID to call a function in hardware. When an HWIP performs a function call, its HWID and its currently executing state are combined to make a return address for the callee.

In this paper, we only consider the software address resolution in case that the software code is statically linked, which is the common case in embedded systems. As shown in Fig. 1, the software and hardware compilation should be tightly coupled; the hardware compilation uses two results from the software compilation process: a symbol table and a linking log file. The symbol table file is extracted from the executable code generated by the software compilation. The table contains address information of all global/static variables as well as software function addresses. The linking log file is used to resolve addresses of software constant data. If a constant like the ("sum =%d") string in Fig. 1 is not declared as a variable, its address will not appear in the symbol table. Since only a linker knows the addresses of constant data, we modified the GNU linker to print out the size and offset information of constant data in object files to the linking log file so that GCC2Verilog can use it to calculate the data addresses. Using the linking log file and the symbol table file as inputs, the hardware compilation can resolve all kind of addresses in software as if it is linked with software code. For the address resolution when the software is dynamically linked, we can apply the same method as described in [24.

V. Verilog Code Generation

One HWIP generated by the GCC2Verilog compiler consists of one datapath and one control unit. The control unit performs the FSM state transition based on control flow and interaction with other components (PICO, other HWIPs or the memory). The control unit controls the datapath execution through a state variable (pc) which is similar as a program counter in a microprocessor. The datapath is generated from RTL instructions and performs the calculation of an HWIP.

Figure 3 shows an example of how the C function is translated into Verilog code using our compiler system. In Fig. 3(a), the example C code contains a *calculate()* hardware function, which is called from the *main* function in software. The HW function uses three pointer variables: *history*, *coefficient*, and *Out* whose contents are dynamically allocated at runtime by the *main* function. In general, the code containing dynamic allocated pointers like this cannot be supported by other HLL-to-HDL translators; however, it is no longer a problem in our approach. The memory is allocated in software, and the allocated address can be loaded from its pointer address obtained through software address resolution. The HW function's RTL code contains four basic blocks (BBs), and they are numbered from 2 to 5. The control flow between BBs and the FSM states contained within each BB are shown in Fig. 3(b); a start and a finish state are inserted at the beginning and the end of the function. A control unit and a datapath of the translated Verilog code are shown in Figs. 3(c) and 3(d), respectively. In this example, the datapath uses 2-port memory.

1. Code Generation for Datapath

The HW instruction scheduler of GCC2Verilog divides the RTL sequence within a BB into FSM states based on data dependences as well as resource constraint between instructions so that independent statements are grouped into one FSM state.

The communication between software functions and GCC2Verilog generating HWIPs follows the PICO's calling convention. Therefore, an HWIP gets its input arguments

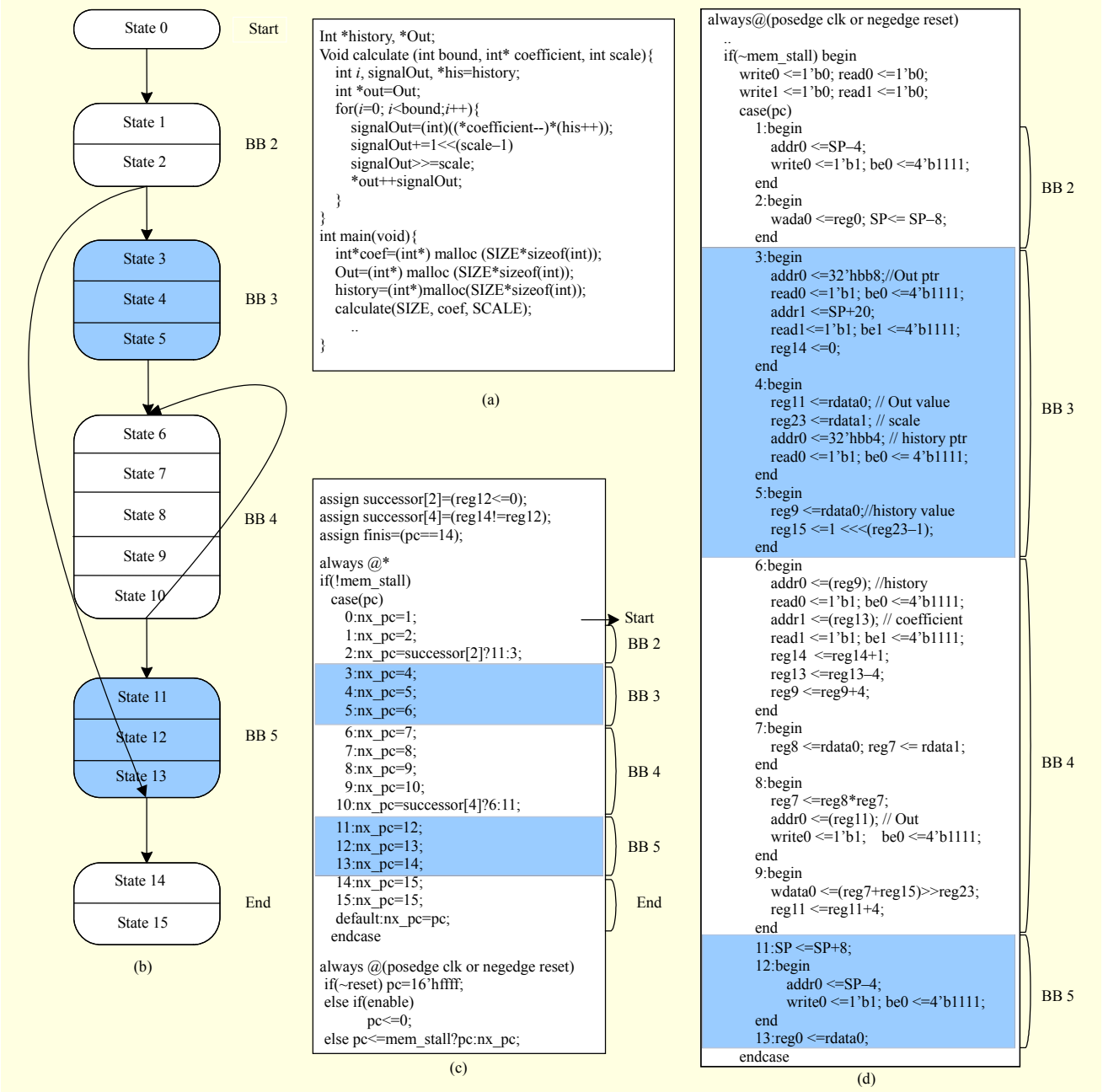

Fig. 3. GCC2Verilog's code generation method: (a) C code, (b) RTL control flow of calculate HW function, (c) control unit, and (d) datapath of generated Verilog module.

through argument registers and a shared stack space with the software. At the entry block, an HWIP pushes a link register (reg0) to the stack. If any caller-saved registers are used inside the HWIP, these registers are also pushed to the stack. After that a stack pointer value is adjusted as shown in state 2. Through the SW address resolution, the hardware knows that the addresses of *Out* and *history* are 0xbb8 and 0xbb4, respectively. At BB 3, the addresses pointed by *Out* and *history* are loaded into reg11 at state 3 and 4 and reg9 at state 4 and 5, respectively. The first two arguments of the *calculate* function (*bound* and *coefficient*) are passed to the HWIP through argument registers, reg12 and reg13; the third argument (*scale*) is passed through a stack. At state 3 of the FSM, the *scale* argument is also loaded into reg23 from the stack address (SP+20). The BB 4 performs the main loop execution: loading elements of the array pointed by *history* and *coefficient*, calculating a value, and writing the value to the memory addresses pointed by *Out*. Finally, the stack pointer SP is restored, and the link register is popped at the exit of BB 5. In hardware, since an arithmetic instruction takes only one clock

Table 3. Description of translated code in EEMBC benchmarks.

Benchmark	HW functions	Called SW functions	Inlined	Line	Unroll	Time (%)
aifirf	t_run_test	th_malloc_x (malloc), GetTestData, th_signal_start, th_exit (exit), GetInputValues, calc_crc32, th_signal_finished	WriteOut	422	3	100
idctrn	t_run_test, GetInputValues	th_malloc_x (malloc), GetTestData, th_signal_start, cos, calc_crc32, th_exit (exit), th_signal_finished	WriteOut, unPack	630, 20	8	100
autcor	fxpAutoCorrelation			20	32	99
conven	convolutionalEncode			45	4	98
fft	fxpfft	assert		102	2	98
bezier	interpolatePoints		paramatric	55	4	99
dither	ditherImage	memset		115	4	99
rgbhpg	t_run_test	th_signal_start, th_signal_finished, calc_crc8		94	16	100
rgbyiq	t_run_test	th_signal_start, th_signal_finished, th_printf (printf), th_malloc_x (malloc), calc_crc8, th_exit (exit)		132	12	100

cycle and the non-blocking assignment is used, we can ignore the anti-dependency among them. For example, at state 6 of Fig. 3(d), the reg9 is read by the (addr0 <= reg9) instruction and updated by the (reg9 <= reg9 + 4) instruction, but the two instructions still can be executed within one FSM state.

2. Code Generation for Control Unit

The control unit of HWIP is implemented with two Verilog *always* blocks as shown in Fig. 3(c). One *always* block (top in Fig. 3(c)) specifies the next state (nx_pc) statically, and the other block (bottom in Fig. 3(c)) determines the next state dynamically, that is, take a current state or the next state. The state transition is decided based on the HW function's internal control flow, control signals from other HWIPs or PICO, and ready signals from memory. An HWIP starts when it receives an *enable* control signal, and stops when it reaches an ending state.

Almost all RTL instructions are executed by the datapath except comparison and jump instructions. A jump instruction changes the control flow among BBs, which means changing the state variable (pc), so it is implemented in the control unit instead of the datapath. A compare instruction is also excluded from the datapath due to the similar reason. The implementation of a branch at state 2 of Fig. 3(c) can be explained as the comparison instruction checking the branch condition implemented as a statement (assign successor[2] = (reg12 <= 0)). The successor[3] signal is then used to perform the jump instruction: (2: nx_pc = successor[2] ? 11 : 3).

A state transition is performed if and only if the memory is not busy, that is, a *mem_stall* signal sent from the memory is not set. The condition ensures that if an FSM state containing memory instructions, the FSM cannot change to another state before the memory instructions finish (pc <= mem_stall ? pc : nx_pc), so that ensuring the correctness of the program. At the end of the function, the control unit announces its finish status to its caller by setting a *finish* signal (assign finish = (pc==14)). After that the HWIP is stalled (15: nx_pc = 15).

VI. Performance Evaluation

In order to evaluate the quality of our generated Verilog code, we used several EEMBC benchmarks [25]. We simulated the Verilog code with the PICO microprocessor on the ModelSim [26] simulator. Also, we used 1, 2, 4, and 8-port memory access configurations for performance evaluation since the number of concurrent memory accesses affects the overall performance significantly.

Table 3 describes in detail the selected codes. The HW functions column lists functions which are translated into Verilog by our GCC2Verilog. The Called SW functions column names the SW functions called by functions in the HW functions column. The inlined functions column lists functions that are inlined within the HW functions, and the Lines shows the number of C code lines of each HW function (including the codes of inlined functions). To exploit ILP, we applied the loop unrolling to the benchmarks; the number of unrolling times was selected for the best performance and it is shown in Unroll column. Both the GCC2Verilog and the PICO compiler were implemented based on the GCC version 4.2.2 [16]. All codes were compiled with –O3.

Figure 4 shows the speedup of PICO+HWIP execution with respect to the PICO-only execution when using 1, 2, 4 or 8-port memory. The speedup gain by PICO+HWIP is 8, 12, 17, and

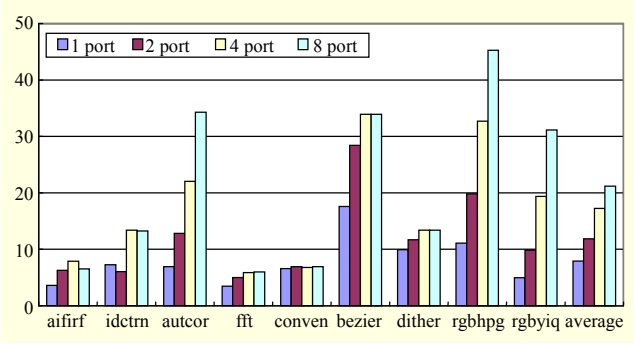

Fig. 4. Speedup of PICO+HWIPs execution with respect to PICO-only.

21 times on average, respectively. The measured execution time of each benchmark is started from *th_signal_start()* to *th_signal_finished()* functions; the two functions are called within the *t_run_test()* function; this is the standard time measurement of EEMBC. The ratios of the execution time of translated functions in HW functions column and their callees to the total execution time are shown in the time column of Table 3. The speedup includes the communication overhead between PICO and HWIPs. We used the approach in [24] for the communication between the PICO and HWIPs. However, the PICO's interface and ISA were optimized so that the SW can call to HW with one machine instruction. Therefore, the calling overhead is very insignificant and almost does not affect the final performance.

In memory intensive benchmarks such as *autcor*, *rgbhpg*, and *rgbyiq*, the speedup is nearly proportionally increased when the number of memory ports is increased. The speedup on *aifirf*, *fft*, and *conven* benchmarks is less significant because loops inside the benchmarks are small, not computation intensive, or containing only few numbers of iterations.

The performance of the Verilog code translated from machine RTL is still unimpressive because the instruction parallelism in RTL is limited. However, we have not yet aimed at the high performance from exploiting various levels of parallelism like in many previous works [2]-[5],[7],[15],[19], but focused on supporting a complete, unmodified translation of a high level language into Verilog with acceptable code quality as the first step of our research. We believe that the parallelism exploitation in FPGA would be similar to or same as the SW parallelism. Therefore, several techniques which are already available in the GCC compiler for higher parallelism exploitation, such as aggressive instruction scheduling for wide-issue superscalar processors and VLIW machines, predicate execution, or the OpenMP for multithread execution, can be applied to improve the quality of our code generation. However, since the FPGA target has some advantages over microprocessors. The FPGA is not limited to predefined instruction set architectures or fixed number of functional units and registers, etc. Our GCC2Verilog may have more opportunities than general SW compilers in optimizing code and exploiting parallelism.

VII. Conclusion

The translation of HLL into HDL for hardware design has not been fully supported due to the large difference between software and hardware programming concepts. This paper proposed a novel approach to build an HLL-to-HDL translator, called the GCC2Verilog compiler based on the commonly used GCC compiler toolsets. By taking the approach, we could reduce the implementation effort and resolve all syntax issues in the translation while providing high performance at the same time. We fully described the method to translate RTL into Verilog code and introduced a complete address resolution method between software and hardware. To the best of our knowledge, our work is the first attempt of making a compiler system and its execution framework which supports the complete translation of C language into the Verilog HDL. Our GCC2Verilog is a cross compiler of GCC, which targets the FPGA instead of microprocessor architectures. We verified the compiler code generation with several EEMBC benchmarks which have complex use of pointers and function calls. The achieved speedup was up to 34 times and 17 times on average with 4-port memory system with respect to the PICO microprocessor.

Even though our system delivers good performance, we need to resolve two main issues. One is to optimize the memory access to achieve good performance even with limited number of memory ports. The other problem is how to support dynamic program execution status in hardware codes like an overflow detection of a signed integer addition in a microprocessor. We leave these issues for our future work.

References

[1] D. Andrews et al., "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24, no. 4, Jul./Aug. 2004, pp. 42-53.

[2] T.J. Callahan, J.R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Comput.*, vol. 33, no. 4, 2000, pp. 62-69.

[3] S.C. Goldstein et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Int. Symp. Comput. Architecture*, May 1999, pp. 28-39.

[4] A. Hormati et al., "Optimus: Efficient Realization of Streaming Applications on FPGAs," *Int. Conf. Compilers, Architectures, Synthesis for Embedded Syst.*, Oct. 2008, pp. 41-50.

[5] "Impulse Tutorial: Generating HDL from C Language," Impulse Accelerated Technology, Inc., 2009.
[6] S. Guptaet al., "SPARK: A High-level Synthesis Framework for Applying Parallelizing Compiler Transformations," *Int. Conf. VLSI Design*, Jan. 2003, pp. 461-466.
[7] J. Villarreal et al., "Designing Modular Hardware Accelerators in C with ROCCC 2.0," *Field Programmable Custom Comput. Mach.*, May 2010, pp. 127-134.
[8] I. Page, "Constructing Hardware-Software Systems from a Single Description," *J. VLSI Signal Process.*, vol. 12, no. 1, 1996, pp. 87-107.
[9] S.A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," *Design, Automation and Test in Europe*, Sept. 2005, pp. 66-67.
[10] A.J. Virginia, Y.D. Yankova, and K.L. Bertels, "An Empirical Comparison of ANSI-C to VHDL Compilers: SPARK, ROCCC and DWARV," *Anual Workshop Circuits, Syst. Signal Process.*, Nov. 2007.
[11] V.V. Sanevelly and R.L. Haggard, "A Procedure for Designing a Translator from C to VHDL," *Southeastern Symp. Syst. Theory*, 2002, pp. 329-333.
[12] D. Soderman and Y. Panchul, "Implementing C Designs in Hardware: A Full-Featured ANSI C to RTL Verilog Compiler in Action," *Int. Verilog HDL Conf. VHDL Int. Users Forum*, 1998, p. 22.
[13] Y.D. Yankova, "DWARV: Delft Workbench Automated Reconfigurable VHDL Generator," *17th Int. Conf. Field Programmable Logic Appl.*, Aug. 2007, pp. 697-701.
[14] R.P. Wilson et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, Dec. 1994, pp. 31-37.
[15] M. Budiu et al., "Spatial Computing," *Int. Conf. Architectural Support Programming Languages Operating Syst.*, 2004, pp. 14-26.
[16] Richard M. Stallman, "GNU Compiler Collection Internals for GCC version 4.2.2," GNU, 2005.
[17] Y. Na et al., "Chip Implementation of PICO Processor," *CCC Conf.*, Feb. 2010.
[18] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," *IEEE Symp. Field Programmable Custom Comput. Mach.*, 1995, pp. 136-144.
[19] Andres Takach, "Catapult C Synthesis: Creating Parallel Hardware from C++," *Int. Symp. Field-Programmable Gate Arrays Workshop*, Feb. 2008.
[20] M.B. Gokhale et al., "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2000, pp. 49-56.
[21] L. Semeria and G.D. Micheli, "Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 20, 2001, pp. 213-233.
[22] E. Anderson et al., "Memory Hierarchy for MCSoPC Multithreaded Systems," *Int. Conf. Eng. Reconfigurable Syst. Algorithms*, June 2007, pp. 44-50.
[23] A.V. Aho et al., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2nd ed., 2006.
[24] G. Nguyen thi Huong and S.W. Kim, "Support of Cross Calls between a Microprocessor and FPGA in CPU-FPGA Coupling Architecture," *Reconfigurable Architecture Workshop*, Apr. 2010.
[25] J.A. Poovey et al., "A Benchmark Characterization of the EEMBC Benchmark Suite," *IEEE Micro*, vol. 29, no. 5, Sept./Oct. 2009, pp. 18-29.
[26] ModelSim, Mentor Graphics Inc. http://www.model.com/

Giang Nguyen Thi Huong received her BS in information technology from Vietnam National University, Hanoi, Vietnam, in 2004. She is now working on her PhD in electrical engineering at Korea University, Seoul, Rep. of Korea. Her research interests include compiler construction, FPGA, and microprocessor architecture.

Seon Wook Kim received the BS in electronics and computer engineering from Korea University, Seoul, Rep. of Korea, in 1988. He received the MS in electrical engineering from Ohio State University, Columbus, Ohio, USA, in 1990, and the PhD in electrical and computer engineering from Purdue University, West Lafayette, Indiana, USA, in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Intel/KSL from 2001 to 2002. Currently, he is a professor and chair with the School of Electrical Engineering of Korea University. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and IEEE.