High-Level Synthesis of Variable Accesses and Function Calls in Software Compatible Hardware Synthesizer CCAP

Masanari Nishimura Kenichi Nishiguchi Nagisa Ishiura
School of Science and Technology
Kwansei Gakuin University
{nishimura.m,scbc9043,ishiura}@ ksc.kwansei.ac.jp

Hiroyuki Kanbara
ASTEM RI
kanbaraastem.or.jp

Hiroyuki Tomiyama
Graduate School of Information Science
Nagoya University
tomiyama@is.nagoya-u.ac.jp

Yutets Takatsukasa Manabu Kotani
Graduate School of Informatics
Kyoto University
{takatsukasa,kotani}@vlsi.kuee.kyoto-u.ac.jp

Abstract—We are developing a high-level synthesis system named CCAP (C Compatible Architecture Prototyper), which synthesizes arbitrary functions in ANSI-C programs into hardware modules callable from the remaining software functions executed on a CPU. The synthesized hardware shares the entire memory space with the CPU and transfers data and controls through global variables. This eliminates the necessity of designing a dedicated interface for each hardware module. Programs including pointers are synthesized in a natural way, so that arrays and dynamic data allocated in the software may be accessed from the hardware using pointers. In this paper, we present the key synthesis techniques employed in CCAP, including the handling of variables, the mechanism of function calls using global variables, and scheduling of the function calls.

I. Introduction

High-level synthesis, which synthesizes register transfer level circuits from behavioral specification of target systems, is gaining popularity as a technology to enhance the efficiency of VLSI design.

Various languages have been used as input to high-level synthesis, including hardware description languages such as Verilog HDL and VHDL, system description languages such as SystemC, SpecC, Bach-C, and Handel-C, and programming languages such as ANSI-C/C++. ANSI-C is an attractive choice among them because of its popularity and fast simulation speed. Moreover, it offers the possibility of exploiting existing programs, which are given as the reference model of the target applications or developed for system validation and performance estimation.

There are a number of high-level synthesis tools which take C/C++ as input, such as SpC [3], Catapult C Synthesis[1], and eXCite[2]. Even though there are some limitations, they can synthesize programs written in C/C++ into hardware. However, these systems are basically designed to synthesize hardware as a single module. Namely, when we design a system consisting of software and hardware, interfaces among software/hardware modules, or how they pass controls and data, are beyond the scope of these systems, and are attributed to a part of system design tasks.

In Japanese patent 2003-114914, a method of synthesizing hardware from behavioral description of systems consisting of hardware and software is proposed. This method enables software to call functions synthesized as hardware by employing a buffer RAM between the CPU and each hardware module which is used to transfer the arguments and the return value. However, since this method does not allow “call-by-reference,” the passing of large objects such as arrays will be extremely inefficient if a small portion of them are referenced or updated. There is also a limitation that the hardware functions cannot call other hardware functions.

We are developing a high-level synthesis tool, named CCAP (C Compatible Architecture Prototyper), which synthesizes arbitrary functions in C program into hardware modules in a way that they can replace the original software functions. The synthesized hardware shares the whole memory space with the CPU, and data and controls are transferred through global variables. This enables software to call hardware functions without the help of extra interfaces. Statements including pointers, “call-by-reference” using pointers are supported as well as function calls from the hardware functions to other hardware functions. In this paper, we propose the key technologies for CCAP, including synthesis of variable accesses, function calls using global variables, and scheduling of function calls.

II. Structure of synthesized system

The synthesizer CCAP transforms a part of the functions chosen by designers in a C program into hardware and executes the others as software on a CPU. Fig. 1 shows the structure of the system synthesized by CCAP. One of the essential points of the system structure is that the synthesized hardware shares the memory space with software.

[1]http://www.mentor.com/
[2]http://www.yxi.com/

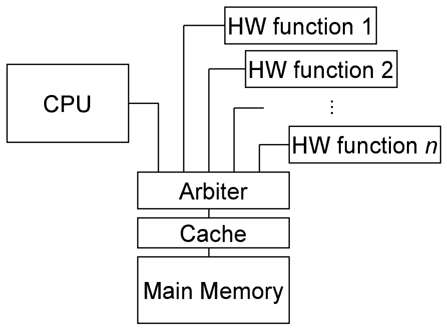


Fig. 1. Structure of synthesized system.

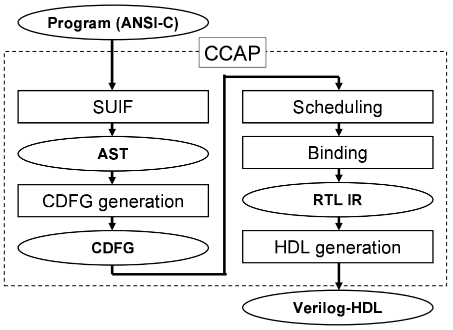


Fig. 2. Flow of synthesis.

The CPU and the hardware modules are connected to the main memory (a cache is optional) through an arbiter.

The arbiter arbitrates the memory accesses from the CPU and the hardware modules. Basically, the memory access requests are processed in order of arrival, but priority is given to the CPU if the requests are simultaneous. The arbiter passes each request to the main memory and raises "stall" signal to the sender until the memory access completes.

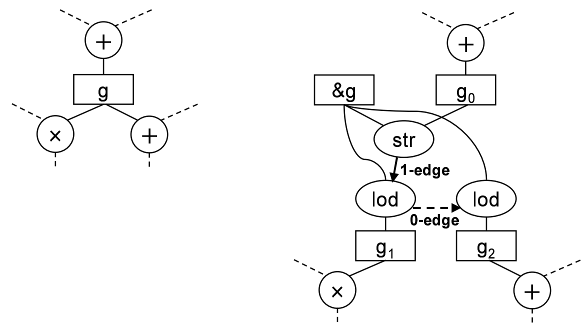
The arbiter is also designed to make the memory accesses efficient. It has write buffers so that the CPU and the hardware modules do not need to wait for the completion of the memory write operation. If read accesses are requested to the same addresses as the recent write accesses, the data cached in the buffers are returned. The arbiter also caches the values of the global variables used for our function call mechanism (discussed in Section V). to avoid concentration of the memory accesses.

Since the hardware and the software can access the identical address space, data can be transferred through global variables. The hardware modules may access data objects allocated by software because the memory accesses using pointers are converted to indirect memory access, which are carried out with the load/store units.

III. Flow of synthesis

Fig. 2 shows the flow of the synthesis in the CCAP. Front-end processing for the input C program such as parsing and hardware-independent optimizations is performed by SUIF[3], and CDFG (Control Data Flow Graph) is generated from the SUIF's abstract syntax tree. The CDFG is then transformed to handle global variables, local arrays, and function calls. After scheduling and binding, the CDFG is converted into a register transfer level intermediate representation, from which a hardware description in Verilog HDL is generated.

[3]http://suif.stanford.edu/



(a) CDFG with global variables (b) Transformed CDFG

Fig. 3. CDFG transformation for global variable accesses.

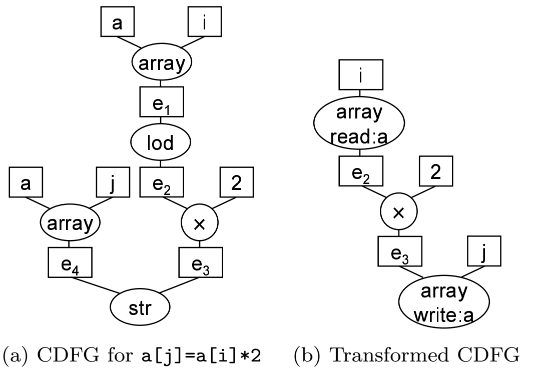
IV. Synthesis of variable accesses

In CCAP, the local variables within the function scopes are mapped to registers, as are in traditional high-level synthesis systems. On the other hand, the global variables are allocated to the main memory so that both the CPU and the hardware modules can access them. The arrays declared global are also allocated to the main memory, while the arrays local to the functions are synthesized into register files to accelerate the accesses.

A. Global variables

We assume that all the global variables shared by the CPU and the hardware modules are *volatile*. Namely, the value read or written to by a certain access is not always equal to that of the next read access. This is because multiple hardware and software processes may access an identical global variable simultaneously and because DMA or memory mapped I/O may be implemented.

We transform the accesses to the global variables into load/store operations to the main memory. Fig. 3 (a) shows a CDFG before transformation (the CDFG generated from an intermediate representation of SUIF). The result of addition is assigned to a global variable g, which is used by succeeding multiplication and addition. At this point, there



(a) CDFG for `a[j]=a[i]*2` (b) Transformed CDFG

Fig. 4. CDFG transformation for local array accesses.

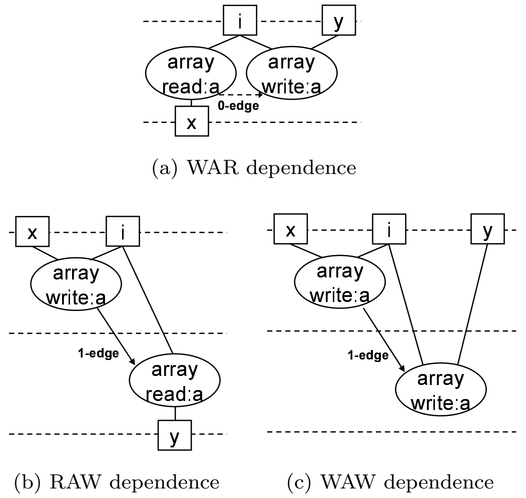


Fig. 5. Simultaneous execution of register file accesses.

is no distinction between the global variables and the local variables in terms of the data structure. Fig. 3 (b) shows the transformed CDFG, where `&g` is the address of `g`, and $\mathtt{g}_0$, $\mathtt{g}_1$, and $\mathtt{g}_2$ are the values of `g` (which are assigned to registers and treated in the same way as the local variables). `Str` and `lod` are store and load operations, respectively. If there are multiple references to a global variable, we insert a load operation for each reference, for we assume the global variables are volatile.

We set dependency edges among the inserted `lod`/`str` operations, which are depicted by the arrows in Fig. 3 (b). We use two kinds of dependency edges; a *0-edge* and a *1-edge*. The 0-edge allows the scheduler to schedule two operations in the same cycle, while the 1-edge forces the subsequent operation to be scheduled to a cycle after the preceding operation. In Fig. 3 (b), the arrow with dashed line is a 0-edge, and the arrow with solid line is a 1-edge. The dependency edges are set based on the order of the appearances of the corresponding variable accesses in the source program. We set a 0-edge between `lod` and `lod`, and 1-edges for the other combinations.

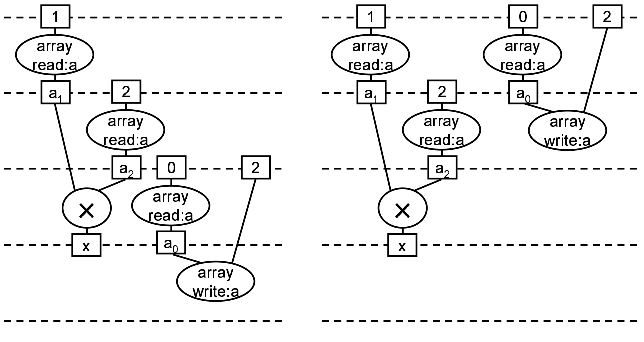
The addresses of the global variables are obtained from executable module generated by a linker. During the hardware synthesis process, the addresses of the global variables are represented in the form of symbolic strings until the register transfer level intermediate representation is converted into the Verilog HDL form, where the symbolic strings are replaced by the actual addresses.

B. Local arrays

Local array variables are mapped to register files. Basically, one register file is allocated to each local array. Fig. 4 gives an example of CDFG transformation associated with local array accesses. In Fig. 4 (a), an `array` operation calculates the address of the element of the array from its base address and the index value, which is used by subsequent `lod`/`str` operations. The sequence of `array` and `lod`/`str` operations is transformed into an operation dedicated to register file accesses as illustrated in Fig. 4 (b). `Array_read:a` and `array_write:a` operations take the index value as input and reads and writes to the register file for array `a`, respectively.

```
x = a[1]*a[2];
a[2] = a[0];
```

(a) program



(b) Single port (c) Multi-port (2R1W)

Fig. 6. Scheduling of register file accesses.

We assume in our system that the register files may have multiple ports so that they can transact multiple accesses in a single cycle. Fig. 5 shows the dependency edges that must be introduced for correct scheduling. We assume that the read accesses to the register files are asynchronous and the write accesses are synchronous. In the case of WAR dependence, the both operations can be executed in the same cycle because `array_read` reads the data during the cycle while `array_write` update the register file at the end of the cycle. In the case of RAW and WAW dependence, the subsequent operations must be delayed more than one cycle. Fig. 6 gives an example of scheduling; the program in Fig. 6 (a) is scheduled as in Fig. 6 (b) with a single port register file, as in Fig. 6 (c) with a multiple ports (2 read/1 write) register file.

```
void caller() {                  void caller() {
   ...                              ...
   b=callee(e1,...,en);     ⇒       _ARG_callee_1=e1;
   ...                              ...
}                                   _ARG_callee_n=en;
                                    _RUN_callee=1;
                                    WAIT(_RUN_callee);
                                    b=_RET_callee;
                                    ...
                                 }
```

Fig. 7. Caller side transformation.

```
int callee(a1,···,an){            int callee(a1,···,an){
   body                     ⇒       while(1){
   return z;                           WAIT(!_RUN_callee);
}                                      a1=_ARG_callee_1;
                                       ...
                                       an=_ARG_callee_n;
                                       body
                                       _RET_callee=z;
                                       _RUN_callee=0;
                                    }
                                 }
```

Fig. 8. Callee side transformation.

C. Comparison and limitation

SpC [3] synthesizes C programs with pointers by means of a tagging technique based on static pointer analysis. It also realizes synthesis of `malloc`/`free` statements by introducing hardware for memory management. However, it can only handle pointers to local variables; pointers to global variables or software allocated objects and passing of arguments by reference are beyond the scope of this method. This may be partly because it aims at designing a single hardware module from a C program. On the other hand, CCAP focuses on pointers to global variables and data allocated in the stack and heap areas so that synthesized hardware may have interoperability with software functions. However, in our current implementation, pointers to local variables cannot be synthesized. Moreover, local structs and local arrays whose sizes are not determined at compile time cannot be handled. We are planning to cope with these data structures by allocating them to the stack area.

V. Synthesis of function calls

In this section, we propose a mechanism of calling hardware functions using global variables. We also propose a new algorithm of the scheduling of function calls which takes the latencies of the function execution into account.

A. Function calls using global variables

Suppose we are synthesizing a function "callee" into hardware. We introduce three classes of global variables associated with this function:

1. `_RUN_`*callee*
   Used for the control of starting and finishing of function "callee." Its value is either 0 (meaning "not in execution") or 1 ("in execution").

2. `_ARG_`*callee*`_1`, ... , `_ARG_`*callee*`_n`
   Used for passing the values of the arguments.

3. `_RET_`*callee*
   Used for passing the return value.

We transform a function call statement into a series of assign statements on these global variables.

Fig. 7 shows the transformation of the statement to call *callee* from *caller*. First, the values of the arguments $\mathtt{e}_1$, ..., and $\mathtt{e}_n$ are assigned to the corresponding global variables `_ARG_`*callee*`_1`, ..., and `_ARG_`*callee*`_n`. Then, value 1 is set to `_ARG_`*callee* to indicate to *callee* to start execution. Since the value of `_RUN_`*callee* remains 1 until it is turned off by *callee*, *caller* waits for `_RUN_`*callee* to be 0 with the WAIT operation. Finally, the return value is received via `_RET_`*callee*.

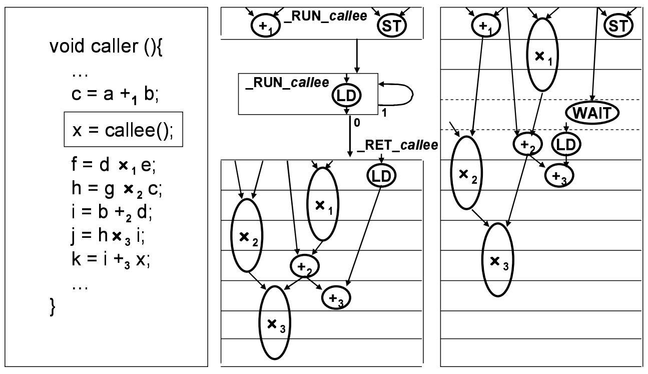
There may be many ways to implement the WAIT operation in Fig. 7 including the use of interrupts. However, we assume the busy loop implementation in this paper for simplicity, for it requires no extra hardware or software. It means to transform WAIT to a C code "`while (_RUN_`*callee*`) {}`." However, as we will explain shortly, we handle WAIT in hardware function as a single special operation so as to improve flexibility of scheduling.

Fig. 8 shows the transformation on the *callee* side. Function *callee* idles until the value of `_RUN_`*callee* is bring up to 1 and it starts the execution. *Callee* receives the value of the arguments by copying from the global variables `_ARG_`*callee*`_1`, ..., and `_ARG_`*callee*`_n` to the local variables $\mathtt{a}_1$, ..., and $\mathtt{a}_n$. After the calculations in the body, the return value is set to `_RET_`*callee* and then 0 is assigned to `_RUN_`*callee* to exit the function.

B. Comparison and limtation

[4] has also proposed a method of synthesizing a part of functions in a source program into hardware that can be activated from software. However, it uses a dedicated buffer RAM between the CPU and each hardware module to transfer the arguments and the return value. While this ensures exclusive data exchange between the CPU and each hardware module, passing of arguments by reference is impossible. Moreover, a hardware function cannot call other hardware functions in this method. These limitations may force designers to rewrite programs or even redesign the overall program structures.

In contrast, CCAP enables data transfers using pointers. This is especially efficient when only small parts of large data structures are referenced or updated. Accesses to dynamically allocated data and hardware function calls from hardware function are also supported. However, recur-



```
void caller (){
  ...
  c = a +1 b;
  x = callee();
  f = d ×1 e;
  h = g ×2 c;
  i = b +2 d;
  j = h ×3 i;
  k = i +3 x;
  ...
}
```

(a) Source program (b) `while` statement (c) WAIT operation

Fig. 9. Scheduling of function call.

s_0 : w = LSU.load(&_RUN_*callee*)
if (w) $\rightarrow s_0$ else $\rightarrow s_1$
s_1 :

Fig. 10. State transition of WAIT operation.
(LSU represents load/store unit.)

sive function calls and software function call from hardware function are not allowed in the current framework.

C. Scheduling of function calls (WAIT operations)

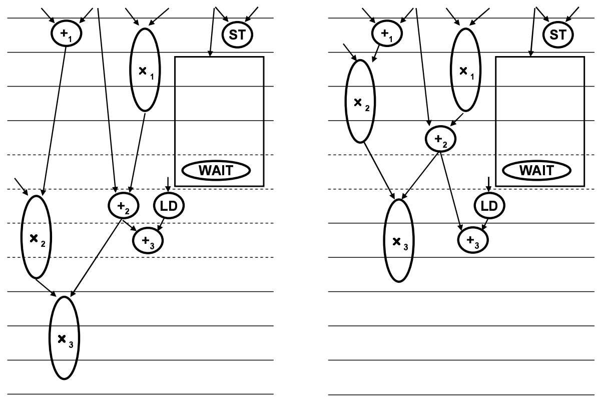
When a hardware function calls another hardware function, the caller function must wait for the completion of the callee function with special operation WAIT. The callee function also must wait until it is indicated to start execution.

Consider the source program in Fig. 9 (a), where 3 operations $\times_1$, $\times_2$, and $\times_3$ are independent of the arguments and the return value of the function. If we simply replace the WAIT operation by a `while` statement, the CDFG is broken into three basic blocks as shown in Fig. 9 (b). On the other hand, if we handle it as a single operation, we will have a larger basic block and higher flexibility of scheduling (Fig. 9 (c)).

The WAIT operation and the associated load/store operations (to pass the arguments, the return value, and the control) can be scheduled in any cycles between the preceeding and succeeding memory access operations.

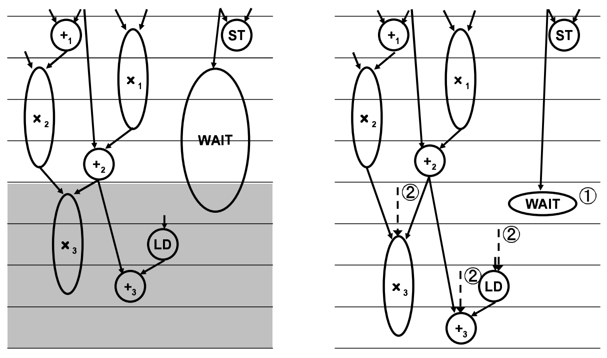
The WAIT operation is finally, at the RTL generation phase, expanded to the state transition in Fig. 10, which is equivalent to the `while` loop implementation. The scheduling for the WAIT operation needs some care because it is a multi-cycle operation whose latency is indefinite; it must be assigned to an exclusive step where no other operations are executed.

Although the number of execution cycles of the callee function is generally indefinite at synthesis time, we can tell the lower bounds of it from the scheduling result of the callee function. This fact is capitalized to further improve



(a) Scheduling equivalent to Fig. 9 (c) (b) Optimized scheduling

Fig. 11. The case when lower bound of length of execution cycles of function is known.



(a) Result of list scheduling (b) Adjustment

Fig. 12. Scheduling algorithm for WAIT operation.

the scheduling of the caller function. For instance, suppose the minimum latency of function *callee* is 4. Then the scheduling in Fig. 9 (c) is equivalent to that in Fig. 11 (a). We can reduce the entire execution cycles by moving $\times_2$ to the idle steps, as shown in Fig. 11 (b).

As a method of finding a scheduling like this, we propose in this paper a framework that can utilize any existing algorithm as a scheduling engine, instead of developing a dedicated new algorithm. The outline of the strategy is as follows:

1. Schedule the given DFG by an arbitrary scheduling algorithm, regarding the WAIT operations as multi-cycle operations with the latencies equal to the lower bounds of the cycles of the corresponding callee functions. (Fig. 12 (a) shows an example of the scheduling result.)

2. Then, change each WAIT operation back to a single cycle operation and place it at the last step of it seen as the multi-cycle operation. (Fig. 12 (b) ①.)

TABLE I
Results of synthesis (preliminary).

		Cycles	Registers
convolution.c	LDST	41	20
	RF	30	21
fir.c	LDST	96	38
	RF	28	18
c_fir.c	while	49	26
	wait	40	22

ADD×2, ALU×2, MUL×1, LDST×1

3. Adjust scheduling by 3. (a) and 3. (b) so that an exclusive execution cycle is secured to each WAIT operation (Fig. 12 (b) ①).
   (a) Mark the operations that finish later than the WAIT operation. (All the operations, except for WAIT, which have their any portion in the shaded region in Fig. 12 (a) are marked.)
   (b) Increase the start cycle of the marked operations by *s* steps, where *s* is the maximum length from the start cycles of the marked operations to that of the WAIT operation.

VI. Experiments and Results

The CCAP synthesizer has been implemented with Perl (ver. 5.8.7) and runs on Linux or on the Cygwin environment. We use SUIF (ver. 1.3.0.1) as the front-end. We use list scheduling algorithm as the scheduling method, and greedy algorithm as the binding method.

TABLE I shows some preliminary results with `convolution.c` and `fir.c` from DSPstone[4]. They are synthesized under the resource restriction of 2 adders, 2 ALUs, 1 multiplier, and 1 load/store unit where all the operations take 1 cycle except that multiplication takes 2 cycles. The loops in the source program are unrolled. The row "LDST" shows the result where the local arrays are allocated to the main memory while "RF" shows the result where they are synthesized into register files. The major reason of the fewer execution cycles in RF is the elimination of address calculations. `c_fir.c` calls functions twice whose minimum length of the execution cycle is 4. The row "while" shows the result where WAITs for the function calls are transformed into `while`s, and "wait" shows that they are scheduled as operations. The scheduling method we proposed reduces the length of entire execution cycles.

VII. Conclusion

We have presented a method of handling of variables, function calls using global variables, and scheduling of function calls in high-level synthesis system CCAP. CCAP makes it possible to replace software function with synthesized hardware and to share data among software and hardware modules using pointers.

We plan to relax the restrictions of behavioral description and evaluate the performance of synthesized hardware.

[4]http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html

Acknowledgement

We would like to thank N. Umehara and T. Nakatani of Ritsumeikan University, and the members of Ishiura Laboratory of Kwansei Gakuin University for their discussion and suggestions.

References

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

[2] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura: "A C-Based Synthesis System, Bach, and its Application," in *Proc. ASP-DAC 2001*, pp. 151–155 (Jan. 2001).

[3] L. Séméria, K. Sato, and G. De Micheli: "Synthesis of Hardware Models in C With Pointers and Complex Data Structures," in *IEEE Trans. VLSI Systems*, vol. 9, no. 6, pp. 743–756 (Dec. 2001).

[4] Kazuhisa Okada: "Software/Hardware Co-design Method," Japanese patent 2003-114914 (2003).