Calling Software Functions from Hardware Functions in High-Level Synthesizer CCAP

Masanari Nishimura Nagisa Ishiura Yoshiyuki Ishimori

School of Science and Technology
Kwansei Gakuin University
{nishimura.m,ishiura,ishimori}@ksc.kwansei.ac.jp

Hiroyuki Kanbara

ASTEM RI

kanbara@astem.or.jp

Hiroyuki Tomiyama

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

Abstract—We are developing a high-level synthesizer named CCAP (C Compatible Architecture Prototyper), which synthesizes functions in C programs into hardware modules that are callable from the other software functions. In this paper, we propose a novel framework where the synthesized hardware functions can call software functions. We give both multi-thread and single-thread implementation schemes. We verified the correctness of the proposed method (single-thread version) through register transfer level simulation.

I. Introduction

Recently, high-level synthesis has become an essential technology to enhance the efficiency of VLSI design which are growing both in scale and in complexity. Input behavioral specifications to high-level synthesis are given in various languages, including hardware description languages (Verilog HDL[1], VHDL[2], etc.), system description languages (System C[3], SpecC[4], etc.), and programming languages (C, C++, etc.). C/C++ is an attractive choice for its rich development environments and fast simulation speed. It also offers an opportunity to exploit existing programs, which are given as reference models of the target applications or are developed for system validation and performance estimation. Although there are a number of high-level synthesis tools which take C/C++ as their input, they are mainly designed to synthesize individual hardware modules and designers have to design interface between software and hardware modules.

We have been developing a high-level synthesis tool named CCAP (C Compatible Architecture Prototyper), which takes ANSI C programs as input and synthesizes arbitrary functions in the program into hardware modules [1][2]. Since the synthesized hardware modules are callable from other functions executed on a CPU, they are referred to as "hardware functions" while the functions executed on the CPU are called "software functions." The software functions and the hardware functions share the entire address space so that they can transfer controls and data through global variables placed on a main memory. This framework enables interaction between software and hardware without designing extra interfaces.

One of the major features of CCAP is that the synthesized hardware functions are callable from both software functions and the other hardware functions. It is also an advantage of CCAP that the software/hardware functions may pass pointers as arguments and return values. Thus CCAP can synthesize broader class of C programs than the other high-level synthesizer. However, calling of software functions from hardware functions has been a challenge, which enables synthesis of C codes containing library/systems calls such as `malloc`/`free`, or calls to "hard-to-synthesize function."

This paper proposes a novel scheme that enables synthesized hardware functions to call software functions. We show two ways, a multi-thread version and a single-thread version, of implementing this scheme. In a preliminary experiment by register transfer level simulation, a hardware function successfully performed a task of bucket sort by calling a software function to allocate memory space dynamically.

II. Related works

There are a number of high-level synthesis tools which take C/C++ as their input, such as Catapult C Synthe-

[1]http://www.eda.org/sv-ieee1800/.
[2]http://www.eda.org/vhdl-200x/.
[3]http://www.systemc.org/.
[4]http://www.cecs.uci.edu/ specc/.

sis[5], eXCite[6], and so on. Since they are basically designed to synthesize individual hardware modules, we need to design extra interface if we want the synthesized hardware modules to interact with software or other hardware modules.

Japanese patent 2003-114914 [3] introduces a dedicated RAM between a CPU and each hardware module so that software functions running on the CPU can call the hardware function by passing arguments and the control via the RAM. This framework, however, does not allow "call by reference," so the efficient data transfer via pointers cannot be synthesized. Calling other hardware functions (not to mention software functions) from hardware functions is also beyond the scope.

SpC [4] synthesizes data accesses via pointers in C programs by static pointer analysis and a tagging technique. It also synthesizes `malloc` and `free` by introducing devoted hardware modules. However, since SpC also handles a single independent hardware module, passing of pointers and sharing of data objects among software/hardware modules are not considered.

On the other hand, CCAP synthesizes hardware modules which share the entire memory space with software functions. This allows the hardware functions to access the data objects allocated by software functions via pointers. The hardware functions as well as the software functions can call the hardware functions.

However, activation of software functions from hardware functions has not been implemented so far in CCAP nor in existing high-level synthesis systems. Calling software functions from hardware functions would extend the applicability of high-level synthesis significantly. For example, C programs including library function calls such as `malloc`/`free` can be synthesized into hardware, in spite that the library functions themselves are very difficult to synthesize. Hardware functions may also call software functions to process error handling and system calls or large complex tasks which are not so frequently executed.

III. High-level synthesizer CCAP and calling of hardware function

Fig. 1 shows the structure of a system designed using CCAP. CCAP synthesizes part of the functions (selected by designers) in a C program into hardware, while the other functions are executed as software on a CPU. The hardware functions and the CPU are both connected to the main memory through the arbiter (cache may be optionally placed between the arbiter and the main memory). The hardware functions share the identical address space with the CPU, thus, the hardware and the software functions can transfer data via global variables on the main memory. Note that pointers as well as normal data can be passed among the software and hardware functions. Data accesses through pointers are synthesized into indirect loads/stores. Hardware functions may access large objects or dynamically allocated objects without rewriting their source C codes.

[5]http://mentor.com/
[6]http://www.yxi.com/

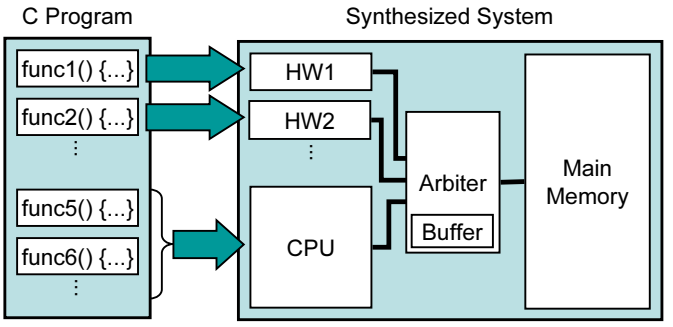


Fig. 1. Structure of a system designed using CCAP.

Calling of hardware functions from software functions are realized by using this data transfer mechanism. The arguments, the return value, and the control are passed via global variables. Fig. 2 illustrates the C-level transformation of a function call in CCAP, where a function `caller` (either software or hardware) calls another function `callee` (hardware). As in Fig. 2 (a), `caller` copies the values of arguments $\mathtt{x_1}$, ..., $\mathtt{x_n}$ to the global variables _ARG_*callee_1*, ..., _ARG_*callee_n*. Then `caller` sets 1 to _RUN_*callee*. It is a global variable to control `callee`: `callee` starts its execution as soon as _RUN_*callee* becomes 1; when `callee` finishes its task then _RUN_*callee* is turned off to be 0. `caller` waits for the _RUN_*callee* to be 0 (WAIT0) and receives the return value through _RET_*callee*. On the other hand, `callee` waits for the value of _RUN_*callee* to be 1 (WAIT1) as in Fig. 2 (b), receives the values of the arguments from _ARG_*callee_1*, ..., _ARG_*callee_n* and then does its task. After assigning the return value to _RET_*callee*, it switches _RUN_*callee* to 0.

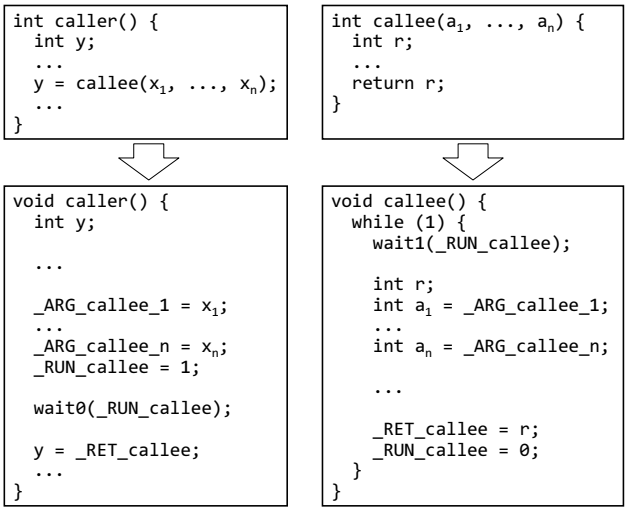
A hardware function may call other hardware functions. However, recursive calls of hardware functions are not supported.

IV. Calling software functions from hardware functions

A. Multi-thread implementation

By using multi-threading, calling software functions from hardware functions can be realized in the same way as calling hardware functions from software functions. Fig. 3 shows the overview of the idea.

We create a thread for each software function at the beginning. Each thread plays a role of the stub for the software function. Then, the thread runs a routine (e.g. `sw1_IF()`, `sw2_IF()`, ...) which performs two tasks to



(a) Caller. (b) Callee.

Fig. 2. Transformation for calling hardware functions.

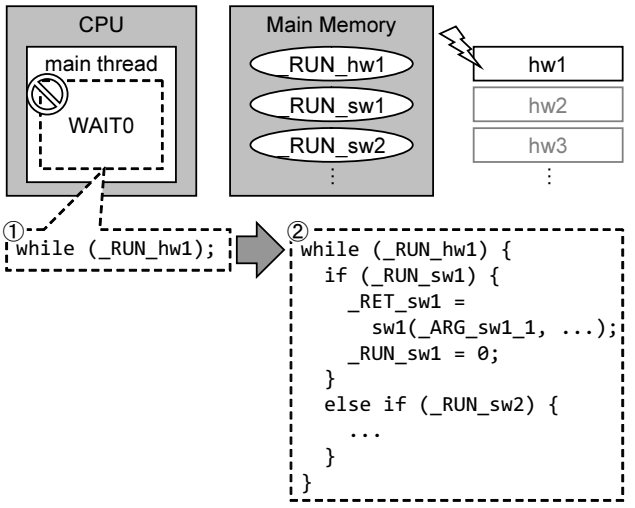


Fig. 4. Single-thread implementation.

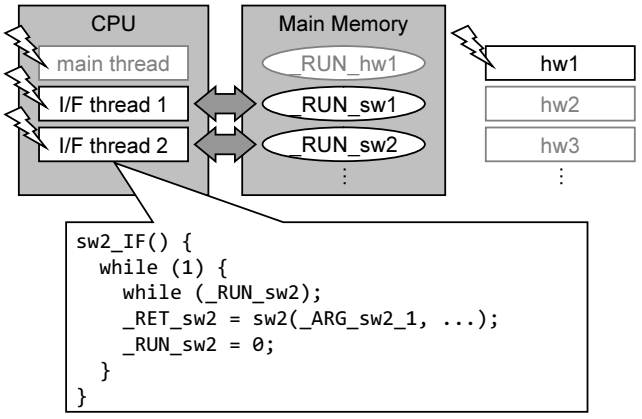


Fig. 3. Multi-thread implementation.

interface with the hardware functions; to watch `_RUN_*` variable of the corresponding software function, and to call the software function when the value of the variable is turned to 1.

The details of the interface routine is as shown at the bottom of the figure. In this routine, the busy loop at line 3 waits for the invocation of `sw2` from the hardware function. When the value is turned to 1, then the software function `sw2` is called from the routine.

B. *Single-thread implementation*

The multi-thread implementation is quite straightforward, but the support for multi-threading by an operating system and a library is a prerequisite. Moreover, execution overhead for watching `_RUN_*` variables would be prohibitive when many software functions can be called from hardware functions. Thus, we propose a single-thread implementation for calling software functions from hardware functions.

The basic idea is to modify the busy loop in the caller function to perform two tasks; to wait for the completion of the callee hardware function, and to pass control to the software function that the hardware function calls. While watching the `_RUN_hw` variable for the hardware function, it also watches the `_RUN_sw*` variables for the software function called by the hardware function. When one of the `_RUN_sw*` becomes 1, it calls the corresponding software function. Figure 4 illustrates the details of the new monitor process. The while loop (①) in the previous calling sequence is extended to include the interfaces (watching and calling) for software functions (②). The loop should contain the interfaces to all the software functions called directly from the hardware functions.

V. Preliminary experiment

We did a preliminary experiment to verify the correctness of our proposed method. Figure 5 shows the outline of the C program used in the experiment, where `main` and `malloc` are executed as software and `bucket_sort` is synthesized into hardware. We used a MIPS R3000 soft-core processor as a CPU. Since C libraries for MIPS R3000 is not available, the `malloc` function was hand-coded. We implemented the single-thread version of the proposed methods and constructed a register transfer level model of the system consisting of the CPU, the `bucket_sort` hardware, an arbiter, and a main memory. The simulation was run using ModelSim (XE II 5.8c) and we verified that the `bucket_sort` module successfully called `malloc` using our scheme and achieved sorting correctly.

```
int main() {  /*SW*/
  initialize an array a[];
  call bucket_sort(a);
}

int* malloc(int size) {  /*SW*/
  allocate a dynamic object;
}

void bucket_sort(int a[]) {  /*HW*/
  calculate the number of buckets k;
  call malloc(k) to allocate the buckets;
  do sorting using the buckets;
}
```

Fig. 5. Outline of experimented C program.

VI. Conclusion

We have presented a method of augmenting the ability of high-level synthesizer CCAP so that synthesized hardware functions can call software functions. We confirmed through hardware simulation that this scheme works well even on an environment where multi-thread is not available. It is one of our project goals to make CCAP accept broader class of C programs so that we can extend the applicability of high-level synthesis. Parallelization of hardware functions is also necessary task to make CCAP more practical synthesis tool.

Acknowledgement

We would like to thank T. Nakatani and N. Umehara (formerly with Ritsumeikan University) for their cooperation in the experiment in Section V. We would also like to thank Y. Sugihara of Kyoto University and the members of Ishiura Laboratory of Kwansei Gakuin University for their discussion and suggestions.

This work is in part supported by KAKENHI 19700040.

References

[1] M. Nishimura, K. Nishiguchi, N. Ishiura, H. Kanbara, H. Tomiyama, Y. Takatsukasa, and M. Kotani: "High-level synthesis of variable accesses and function calls in software compatible hardware synthesizer CCAP," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp. 29–34 (Apr. 2006).

[2] H. Kanbara, T. Nakatani, N. Umehara, N. Ishiura, and H, Tomiyama: "Speed Improvement of AES Encryption using hardware acclererators synthesized by C Compatible Architecture Prototyper (CCAP)," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies* (Oct. 2007).

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

[4] L. Séméria, K. Sato, and G. De Micheli: "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Trans. VLSI Systems*, vol. 9, no. 6, pp. 743–756 (Dec. 2001).