IEICE TRANS. FUNDAMENTALS, VOL.E91–A, NO.12 DECEMBER 2008

LETTER *Special Section on VLSI Design and CAD Algorithms*

High-Level Synthesis of Software Function Calls

Masanari NISHIMURA[†], ***Nonmember***, **Nagisa ISHIURA**[†a)], ***Member***, **Yoshiyuki ISHIMORI**[†], ***Nonmember***, **Hiroyuki KANBARA**[††], ***and*** **Hiroyuki TOMIYAMA**[†††], ***Members***

SUMMARY This letter presents a novel framework in high-level synthesis where hardware modules synthesized from functions in a given ANSI-C program can call the other *software* functions in the program. This enables high-level synthesis from C programs that contains calls to hard-to-synthesize functions, such as dynamic memory management, I/O request, or very large and complex functions. A single-thread implementation scheme is shown, whose correctness has been verified through register transfer level simulation.

key words: *high-level synthesis, CCAP, hardware/software co-design, C-based design*

1. Introduction

High-level synthesis has become an essential technology to expedite the design process of VLSI which are growing both in scale and complexity. Especially, synthesis from ANSI-C/C++ is an attractive choice, when hardware design is having close interaction with software design and when the design activities may even migrate between software and hardware domains.

[1] proposes a high-level synthesis framework focusing on interoperability of software and hardware. The resulting synthesizer *CCAP* enables synthesis of arbitrary functions in a given C program into hardware modules (called *hardware functions*) in the form where they are callable from the other software functions as well as hardware functions in the C program. It also enables the use of pointers and sharing of static/dynamic data objects on a main memory among the software and hardware functions.

Although this framework extends the applicability of the high-level synthesis technology, there are yet many obstacles in the way of synthesizing arbitrary C programs without rewriting. One of them is the call to *hard-to-synthesize* functions. An notorious example is a call to `malloc`/`free` functions. Although SpC [2] has given one solution to this problem, the use of the pointers and the dynamically allocated memory areas are allowed only within a single function. In general, library calls, such as `printf` and math functions are the major hurdles. There have been no universal formula to counter this problem. Another example of functions which are not suitable for synthesis are extremely large complex ones or rarely called ones (such as error handlers). If these functions exist, all the other functions which are ancestors in the call tree become unsynthesizable.

This letter presents an attempt to solve this problem by introducing a mechanism of activating software functions executed on the CPU from hardware functions, which has not been implemented in [1] nor in the other existing high-level synthesizer. We show a single-thread implementation to make this scheme work. 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.

2. High-Level Synthesizer CCAP

Figure 1 shows the structure of a system designed using CCAP. CCAP synthesizes a part of the functions (specified 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 connected to the main memory through an 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. Data access through pointers are synthesized into indirect loads/stores.

Calling of the hardware functions from the software functions is realized by using this data transfer mechanism. The arguments, the return value, and the control are passed

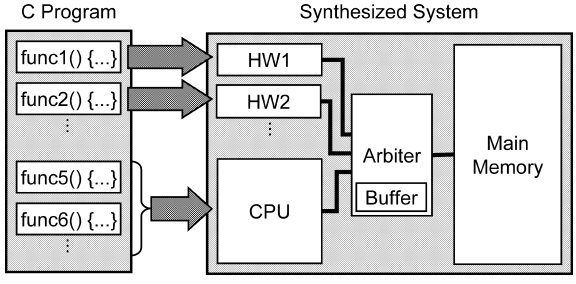

Fig. 1 Structure of a system designed using CCAP.

Manuscript received March 24, 2008.

[†]The authors are with Kwansei Gakuin University, Sanda-shi, 669-1337 Japan.

[††]The author is with Advanced Software Technology & Mechatronics Research Institute of Kyoto (ASTEM RI), Kyoto-shi, 600-8813 Japan.

[†††]The author is with Nagoya University, Nagoya-shi, 464-8603 Japan.

a) E-mail: nagisa.ishiura@ml.kwansei.ac.jp

DOI: 10.1093/ietfec/e91–a.12.3556

via global variables. Calling of the hardware functions from hardware functions is also realized in the same way[†].

3. Calling Software Functions from Hardware Functions

Calling of software functions from hardware functions can be realized basically in the same way as calling of hardware functions. The arguments and the return value can be passed via the main memory. However, control transfer from hardware back to the CPU needs some tricks, for the CPU is busy watching for the completion of the hardware function.

We first show an implementation utilizing multi-threading, which is rather a straightforward method, and then present an improved single-thread version.

3.1 Multi-Thread Implementation

If a multi-threading environment is available on the CPU, we can make software functions behave like hardware functions, and thus we can pass control from hardware to software in the same way as in hardware to hardware.

Figure 2 illustrates the outline of the method, where the main function (①) calls a function `hw()` (②), which is synthesized into hardware, and `hw()` calls a hard-to-synthesize function `sw1()` (③), which is executed on the CPU.

The transformation for ② is done as described in [1]: ② is converted into an interface (stub) function (ⓐ), which is executed on CPU, and the body function (ⓑ), which are synthesized into hardware. The interface function stores the argument into shared variable `_ARG_hw` and set `_RUN_hw` to 1 to activate the hardware function. It waits for the hardware to complete the task to turn `_RUN_hw` to 0 and receives the return value from `_RET_hw`. On the other hand, the hardware waits for `_RUN_hw` to be 1 and loads the argument from `_ARG_hw` to start execution. After storing the return value to `_RET_hw`, it switches off `_RUN_hw`.

Calling to software function `sw1()` in ② is expanded into the code sequence similar to the interface function ⓐ. The code sequence is exactly the same as that of the calling to hardware functions, except for the variable names.

For each callee software function, a thread is created. The thread runs an interface routine `sw1_IF()` (ⓒ) which is generated from the callee software function (③). The interface routine handles the passing of data and control, and calls the body function. In this example, `sw1()` is executed in the main thread if it is called from the main routine and in thread 1 if it is called from some hardware functions.

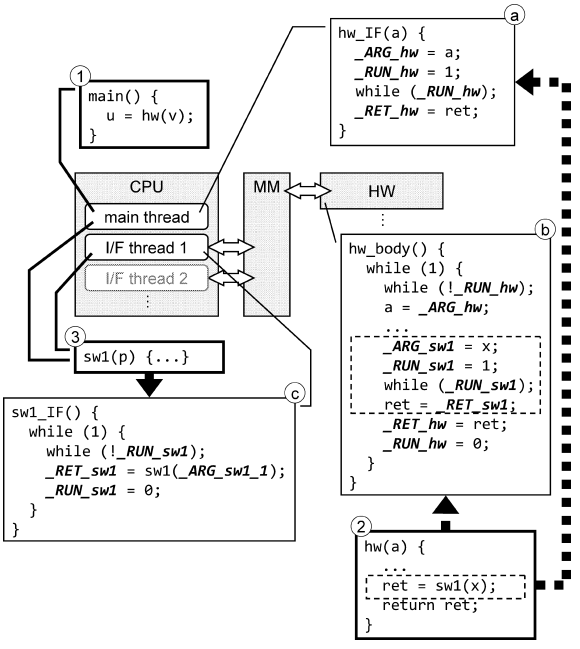

Fig. 2 Multi-thread implementation.

3.2 Single-Thread Implementation

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

The key idea is to integrate all the interface functions associated with each hardware function into a single control management routine. Figure 3 illustrates the overall configuration of the single-thread implementation, where hardware function `hw()` activated from `main()` may call software functions `sw1()`, `sw2()`, and so on.

The control management function `hw_CM()` performs two tasks in a single loop. While it waits for the completion of the hardware body function by watching variable `_RUN_hw`, it also tests `_RUN_sw1`, `_RUN_sw2`, ⋯ to see if

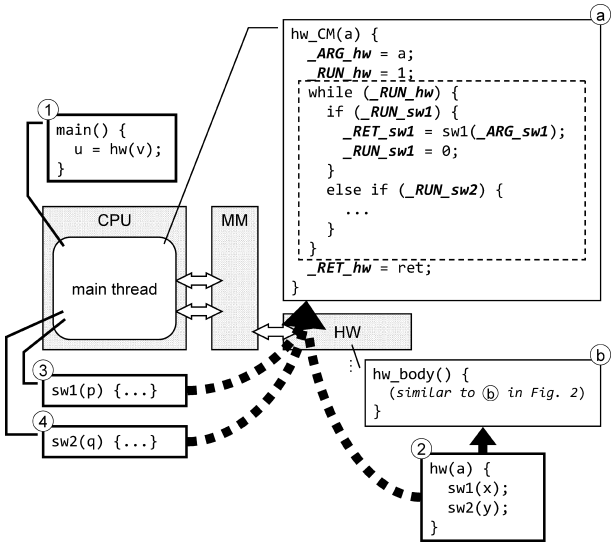

Fig. 3 Single-thread implementation.

[†]Recursive calls of hardware functions are not supported.

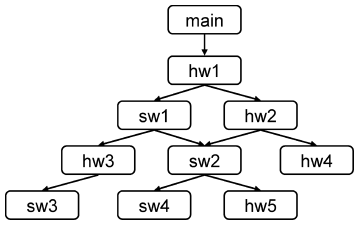

Fig. 4 Example of calling chain.

execution of software functions is requested. On detecting value 1 on any of `_RUN_sw*`, it loads arguments, calls the corresponding body function, stores the return value, and turns the `_RUN_sw*` variable to 0.

The control management functions are necessary for all the hardware functions called from any of the software functions. Note that the while loop in the control management function of a hardware function `hw()` must contain the interfaces to all the software functions that are

- called directly by `hw()`, or
- called by hardware functions in calling chains starting with `hw()` and consisting only of hardware functions.

For example, in Fig. 4, the control management function of `hw1()` must have the interfaces to software functions `sw1()` and `sw2()` (but not necessarily `sw3()` and `sw4()`).

Also note that there is no restriction on the alternation of software/hardware functions in calling chains. For example, calling chain `main()` → `hw1()` → `sw1()` → `hw3()` → `sw3()` in Fig. 4 is well handled in our method.

4. Preliminary Experiment

We did a preliminary experiment to verify the correctness of the 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 `main`. We constructed a register transfer level model of the system consisting of the CPU, the `bucket_sort` hardware, an arbiter, and a main memory and ran simulation using ModelSim (XE II 5.8c). We verified that the `bucket_sort` module successfully called

```
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.

`malloc` and achieved sorting correctly.

5. Conclusion

We have presented a method to synthesize software function calls from hardware functions in high-level synthesis. We confirmed through hardware simulation that this scheme works well.

The proposed method enables synthesis of C codes containing calls to functions which are difficult to implement in hardware. This gives another solution to high-level synthesis of C programs containing pointers and dynamic memory allocation [2]. As high-level synthesis is applied to larger behavioral specifications in variety of areas, it is becoming important that high-level synthesizer can synthesize existing programs without extensive rewriting. It is one of our project goals to make high-level synthesizers accept broader class of C programs.

Acknowledgement

We would like to thank Mr. N. Umehara and Mr. T. Nakatani for their cooperation in the experiment in Sect. 4.

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," Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies, pp.29–34, April 2006.

[2] L. Séméria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.9, no.6, pp.743–756, Dec. 2001.