High-Level Synthesis of Embedded Systems Controller from Erlang

Hinata Takebayashi [1] Nagisa Ishiura [1] Kagumi Azuma [1] Nobuaki Yoshida [2] Hiroyuki Kanbara [2]

[1] Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] ASTEM RI, Kyoto, Japan

Abstract— This article presents a method of specifying the behavior of embedded systems by a subset of Erlang, from which RTL hardware is synthesized. The behavior of the systems is modeled by concurrent processes communicating with each other by message passing. Assembly codes of the BEAM virtual machine compiled from Erlang programs are converted into CDFGs (control dataflow graphs), which are synthesized into Verilog HDL by the back-end of the high-level synthesizer ACAP. Complex routines to handle message passing and garbage collection are synthesized into library modules by the ACAP from reduced C implementation of the BEAM interpreter. A prototype system based on the proposed method implemented in Perl5 has successfully synthesized a simple two-process Erlang program into logic-synthesizable Verilog HDL codes.

I. Introduction

Embedded systems are widely implemented in consumer electronics, automobiles, medical appliances, industrial electronics, etc. In order to meet various needs for these products, higher functionalities and higher performance as well as smaller dimensions and lower power consumption are required to the embedded systems.

An embedded system is commonly implemented as a combination of hardware, processors, and software running on them. When it is difficult to achieve compatibility between performance and power consumption, some functionalities originally implemented as software are migrated to hardware. In order to expedite such re-implementation, various methodologies to automate hardware design based on high-level synthesis [1] have been proposed [2, 5].

However, with the recent rapid development of the network environment and advent of new services and applications utilizing it, networking or coordination of multiple embedded systems is being required. The embedded systems are getting more and more sophisticated in this sense also. It is a next challenge to establish new methodologies to model and to automate design of such communication oriented systems.

Embedded systems can be regarded as a kind of reactive systems which perform certain tasks in response to external events. In order to implement sophisticated controllers which respond to multiple types of events, a modelling based on concurrent processes and interrupts would be necessary. Although real-time operating systems may alleviate the complexity of implementing such systems, high-skills are required to specify interrupt handling and to guarantee response time.

Alternative approach to address this issue is to use domain specific languages which orient event processing and concurrency specification. Erlang [3] is a concurrency oriented functional programming language which is originally developed for implementing telephony switches. Although it is widely used in the area of telecommunications, e-commerce, instant messaging, etc., there are some attempts to use Erlang for embedded systems [4], based on a observation that concurrent processes and message passing will allow succinct description of event processing. If hardware is automatically derived from Erlang specification, advanced sophisticated systems would be easily implemented as efficient devises with higher response and smaller power consumption than processor-based systems.

Thus, this paper proposes a way of specifying the behavior of embedded systems in a subset of Erlang and a method of synthesizing RTL hardware description from the specification. In our modeling, external events arriving at Erlang ports are dealt with by multiple Erlang processes acting in corporation with each other and resulting events are sent out from Erlang ports. In our synthesis method, each Erlang process is implemented as a separate hardware module. Assembly codes of BEAM virtual machine obtained by compiling the input Erlang programs are translated into CDFGs (control dataflow graphs), from which Verilog HDL codes are generated by the back-end of the high-level synthesizer ACAP [5]. Hardware for complex tasks such as message passing or garbage collection, which are difficult to embed into the CDFGs, is generated from reduced C programs of the BEAM interpreter using ACAP.

A prototype synthesis system based on the proposed method is implemented in Perl5, which succeeded in generating Verilog HDL codes from a simple Erlang control program comprising of two processes.

II. Erlang and High-Level Synthesis

A. Programming Language Erlang

Erlang [3] is a concurrency oriented functional programming language originally developed by Ericsson. Erlang expresses concurrency in terms of multiple processes which are generated dynamically and communicating with each other. Information is shared among the processes by message passing, instead of shared variables. The lightweight nature of the processes enables massively concurrent processing of a huge volume of messages.

Fig. 1(a) is a small example of an Erlang program. It defines a function `iprod` which computes the inner prod-

```
-module(iprod).
-export([iprod/2]).

iprod(A,B) -> iprod3(A,B,0).

iprod3([],[],X) -> X;
iprod3([AH|AT],[BH|BT],X) -> iprod3(AT,BT,AH*BH+X).
```

(a) Inner product of lists.

```
01: -module(area_server).
02: -export([start/0, area/2]).
03:
04: start() ->
05:   spawn(fun loop/0).
06:
07: loop() ->
08:   receive
09:     {From, {rectangle, Width, Ht}} ->
10:       From ! {self(), Width * Ht},
11:       loop();
12:     {From, {circle, R}} ->
13:       From ! {self(), 3.14159 * R * R},
14:       loop();
15:     {From, Other} ->
16:       From ! {self(), {error, Other}},
17:       loop()
18:   end.
19:
20: area(Pid, Request) ->
21:   Pid ! {self(), Request},
22:   receive
23:     {Pid, Response} -> Response
24:   end.
```

(b) Area server [3].

Fig. 1. Examples of Erlang programs.

uct of two vectors in the lists. Alphanumeric strings starting with lower case letters are identifiers for functions or atoms (named constants), while alphanumeric strings beginning with upper case letters represent variables. Operator "`->`" defines a function. Erlang variable are of single assignment; each variable is assigned only once. Erlang supports integers of arbitrary precision and floating point numbers, as well as tuples (compound data with a fixed number of elements between "{" and "}") and lists (compound data with a variable number of elements between "`[`" and "`]`").

Fig. 1(b) shows another example which defines a server that computes and returns the area of a given rectangular or a circle [3]. Processes are generated by the `spawn` function, and message send and receive are expressed by the "`!`" operator and the `receive` statement, respectively. When function `start` is called, `spawn` in line 5 generates a process that executes `loop` in lines 7–18 and returns the ID of the process. Function `area` in line 20 takes this process ID and data `Request`, and returns the area of the object in the data. Operator "`!`" in line 21 sends a tuple {`self(), Request`} to the process whose ID is `Pid`. It sends `self()`, its own process ID, together with data `Request` because it wants the server to send the result back. Each process has a queue to receive messages directed to the process. The process takes the messages out of the queue by the `receive` statements as in lines 9–17 and 22–24, which select messages by pattern matching. Timeout may be specified with the `receive` statements optionally.

B. Execution of Erlang Programs

Erlang programs are compiled into byte codes of the BEAM virtual machine, which are executed by a BEAM

```
{function, iprod, 2, 2}.
  {label,1}.
    {func_info,{atom,iprod},{atom,iprod},2}.
  {label,2}.
    {move,{integer,0},{x,2}}.
    {call_only,3,{f,4}}.

{function, iprod3, 3, 4}.
  {label,3}.
    {func_info,{atom,iprod},{atom,iprod3},3}.
  {label,4}.
    {test,is_nonempty_list,{f,5},[{x,0}]}.
    {get_list,{x,0},{x,3},{x,4}}.
    {test,is_nonempty_list,{f,3},[{x,1}]}.
    {get_list,{x,1},{x,5},{x,6}}.
    {gc_bif,'*',{f,0},7,[{x,3},{x,5}],{x,0}}.
    {gc_bif,'+',{f,0},7,[{x,0},{x,2}],{x,2}}.
    {move,{x,6},{x,1}}.
    {move,{x,4},{x,0}}.
    {call_only,3,{f,4}}.
  {label,5}.
    {test,is_nil,{f,3},[{x,0}]}.
    {test,is_nil,{f,3},[{x,1}]}.
    {move,{x,2},{x,0}}.
    return.
```

Fig. 2. BEAM assembly for Fig. 1(a).

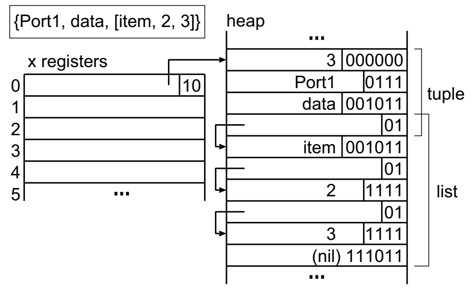


Fig. 3. Internal representation of tuple and list.

interpreter[1]. The BEAM VM has x registers of 1024 words, program counter PC, and stack pointer SP. The Erlang program in Fig. 1(a) compiles to the BEAM assembly code in Fig. 2.

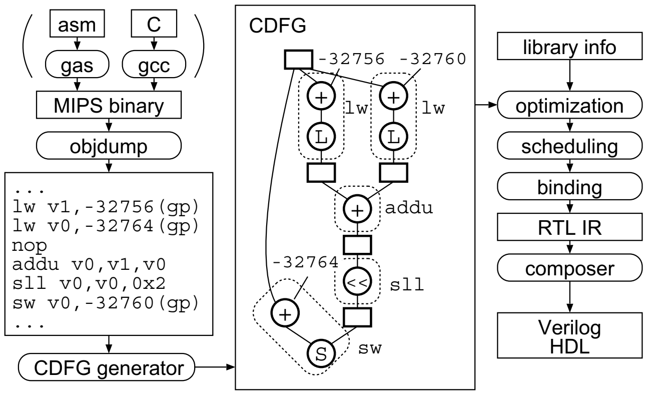
Besides x registers, each process has its own memory storage area. The bodies of aggregate data such as lists and tuples are kept in the *heap* region, while the frame data for functions are stored in *stack* region. If enough space for new data is not available on the heap or the stack, the dead cells in the heap are collected by garbage collection.

The BEAM VM has about 150 instructions, which are roughly classified into the 5 categories:

- Arithmetic and logic operations,
- List and tuple manipulation,
- Jumps and function calls,
- Memory management, and
- Message send/receive.

All the data in Erlang are expressed in terms of the *Eterm*, which can be stored in a 32-bit word. Fig. 3 illustrates the status of the heap when an Eterm to represent {`Port1, data, [item, 2, 3]`} is in register x[0]. A tag to identify the basic type of the Eterm is stored in the lower 2 bits of the word, where b'10' and b'01' stand for a tuple and a list, respectively, and the remaining 30 bits serve as pointers. A tuple with 3 elements consists of 4

[1] There exists a native compiler HiPE which, however, is supported by limited platforms.



```
...
lw v1,-32756(gp)
lw v0,-32764(gp)
nop
addu v0,v1,v0
sll v0,v0,0x2
sw v0,-32760(gp)
...
```

Fig. 4. High-level synthesizer flow in ACAP [5].

words, with 1 word representing the size and the rest for the elements. Each element in a list is expressed using 2 words for its car and cdr. A signed integer with up to 28 bits is represented within a single word, with the lower 4 bits being b'1111.

The message queue of each process is implemented as a linear list. When a message consisting of a single word is sent to the process, a message object is allocated and is appended to the queue. If the message points to an aggregate structure in the heap, the data must be duplicated so that the message can point to the copy. Note that the copy is not directly written into the heap of the recipient process. The copy is first created in a “heap fragment” area which is a newly allocated and linked to the message, and reproduced into the heap of the recipient when it executes a `receive` instruction. Each process has the “current message pointer.” When the process execute the `receive` instruction, the message pointed by the current message pointer (the current message) is copied to x[0] register. If pattern matching succeeds on the message, the message is removed from the queue by a `remove_message` instruction. If not, the current message pointer is advanced by one by `save_message` instruction and the matching is attempted on the next message.

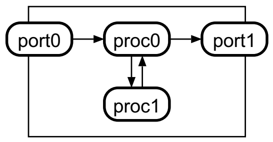
When n bytes are received on an input port, they are delivered to the receiving process as a list of n elements. Contrarily, processes can send a list of byte data to the output port, which is emitted as a byte sequence.

C. *High-Level Synthesis System ACAP*

ACAP [5] is a high-level synthesizer which generates RTL hardware description from C programs or MIPS binary codes. The flow of synthesis is sown in Fig. 4. A binary code generated by GCC or GAS (the GNU assembler) is once disassembled to an assembly code, which is analyzed and translated into a CDFG. Conventional scheduling and binding techniques are applied to the CDFG to generate an RTL code in Verilog HDL.

III. Describing Embedded Systems Control by Erlang

In our method, the behavior of a target system is expressed in terms of multiple Erlang processes. Namely, it is assumed that all the processes are created at the



(a) Communication among processes and ports.

```
-module(roomba).
-export(start/0).

start() ->
  spawn(fun() ->
    register(proc1, self()),
    loop1(0, 0)
  end),
  spawn(fun() ->
    register(proc0, self()),
    Port0 = open_port({spawn, "stdbuf -i0 -o0 -e0 od
-h -w8 /dev/input/js0 | ./controller"}, {packet, 2}),
    Port1 = open_port({spawn, "./roomba"}, {packet, 2}),
    loop0(Port0, Port1)
  end).

decode(Dt,Dh,Et,Eh) ->
  {((Dh bsl 8) bor Dt), ((Eh bsl 8) bor Et)};
decode(X) -> X.

loop0(Port0, Port1) ->
  receive
    {Port0, {data, Data}} ->
      Data2 = decode(Data),
      proc1 ! {proc0, data, Data2},
      loop0(Port0, Port1);
    {proc1, Data3} ->
      Port1 ! {proc0, {command, Data3}},
      loop0(Port0, Port1);
    {Port1, _} ->
      loop0(Port0, Port1);
    _ ->
      loop0(Port0, Port1)
  end.

loop1(D, T) ->
  receive
    {proc0, data, Data} ->
      {Drive, Turn} = calc(Data, D, T),
      Cmd = encode(Drive, Turn),
      proc0 ! {proc1, Cmd},
      loop1(Drive, Turn);
    X ->
      proc0 ! X,
      loop1(D, T)
  end.

calc({Para, X}, Drive, Turn) ->
  if
    X == 258  -> {Para, Turn};
    X == 1026 -> {Para, Turn};
    X == 2    -> {Drive, Para};
    X == 770  -> {Drive, Para};
    true      -> {0, 0}
  end.

encode(Drive, Turn) ->
  if
    Drive =< 57343, Drive >= 32768 ->
      if
        Turn =< 57343, Turn >= 32768 -> {146, 0, 127, 0, 63};
        Turn =< 32767, Turn >= 12288 -> {146, 0, 63, 0, 127};
        true                         -> {146, 0, 127, 0, 127}
      end;
    Drive =< 32767, Drive >= 8192 ->
      if
        Turn =< 57343, Turn >= 32768 -> {146,255,127,255,63};
        Turn =< 32767, Turn >= 12288 -> {146,255,63,255,127};
        true                         -> {146,255,127,255,127}
      end;
    true ->
      if
        Turn =< 57343, Turn >= 32768 -> {146,255,127,0,127};
        Turn =< 32767, Turn >= 12288 -> {146,0,127,255,127};
        true                         -> {146,0,0,0,0}
      end
  end.
```

(b) Behavior description by Erlang.

Fig. 5. Example of Erlang description.

system initialization time and there is no dynamic creation/deletion of processes.

Input/output of the system is performed via Erlang ports, which receive/send byte sequences. Handling of events are regarded as messages passed among Erlang processes and ports. In this paper, communication only

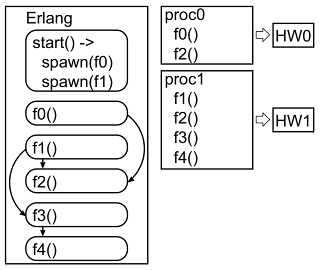


Fig. 6. Hardware modules for Erlang processes.

within the system is handled. Namely, communication via TCP/IP with processes in external systems is out of the scope of this paper. A receiver of a message, which comes on the left-hand side of a "`!`" operator, may be specified in terms of an expression as long as it evaluates to an ID of a process or a port at run time. The data types handled in this paper are limited to 28-bit signed integers, lists, and tuples. Closures are not handled in this paper.

Fig. 5 is a small example of control description by our Erlang subset. The controller receives signals from button presses indicating the directions to move and sends out corresponding signals to the driving devise.

The behavior of the controller can be modeled with two ports and two processes, as illustrated in Fig. 5(a). Port `port0` receives signal data from the buttons and sends control signal data to port `port1`. Process `proc1` just translates the input signal data to the output signal data. Process `proc0` is in charge of data transmission; on receiving data on `port0`, it requests translation to `proc1` and sends the results out via `port1`.

An Erlang code is shown in Fig. 5(b). `start` in line 4 initializes the whole system, creating processes `proc1` and `proc0` in lines 5 and 9, respectively. `proc1` executes the body `loop1` in lines 34–44. `proc0` opens the two ports in lines 11 and 12 and executes the body `loop0` (lines 19–32). Lines 21–24 describe the behavior that when `proc0` receives data from `Port0` it decodes the data and send them to `proc1` for requesting translation. In lines 25–27, the results are sent back from `proc1`, which are forwarded to `Port1`. The other messages are ignored (lines 28–31).

IV. High-Level Synthesis from Erlang

A. Overview

This paper presents a method of synthesizing RTL hardware from control description written in the Erlang subset described in the previous section. In this method, each Erlang process is synthesized into a single hardware module so that processes can run independently of each other except for during interprocess communication. The overhead for scheduling and management of processes are eliminated. The method is this paper assumes that all the data of the processes are placed in a single main memory.

An Erlang process may execute multiple functions. In our method, all the functions for each process, which are recognized by static analysis, are synthesized into a single hardware module. For example, in Fig. 6, processes `proc0` and `proc1` call functions `f0` and `f1`, respectively, and `f0` calls `f2`, and `f1` calls `f2`, `f3`, and `f4`. Then, hardware module `HW0` that implements `proc0` should be able to execute `f0` and `f1`, and `HW1` for `proc1` should execute `f1`, `f2`, `f3`, and `f4`. In this case, a hardware fraction to execute `f2` should appear in both HW0 and HW1.

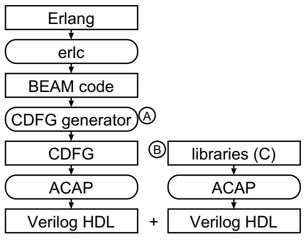


Fig. 7. Synthesis flow of proposed method.

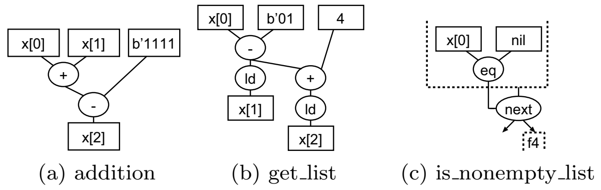


Fig. 8. Conversion from BEAM instruction to DFG.

The flow of synthesis is illustrated in Fig. 7. A given Erlang program is compiled by erlc, an Erlang compiler, into a BEAM assembly code, from which CDFG is constructed. By feeding the CDFG into the back-end of high-level synthesizer ACAP, a Verilog HDL code is obtained. Since some BEAM instructions involve complex tasks such as message passing and garbage collection which would be difficult to embed into the CDFG. In our method, these tasks are implemented as separate hardware modules, called "library modules," which are called from the CDFG. The library modules are synthesized from a reduced C code of the BEAM interpreter by ACAP.

B. Converting BEAM Assembly to CDFG

The BEAM assembly code from erlc is analyzed to create a CDFG for each process. First, the initial process to start the system is scanned to identify all the processes present in the code. Then, all the functions which may be called from each process are enumerated. Each function is decomposed into basic blocks based on branch instructions and target labels. The instructions in each basic block is converted into operations of a DFG (dataflow graph), and finally a CDFG is constructed based on the overall control flow.

BEAM instructions are translated into DFG operations as follows.

(1) Arithmetic and bit operations

Since arithmetic and bit operations in Erlang compiles to `gc_bif` instructions, which execute built-in functions, they are simply converted into operation nodes of DFGs. Since the 28-bit integer data handled in this paper has b'1111 in the lower 4 bits, instructions on them are translated into operation sequences to manipulate the upper 28-bit fields. For example,

```
{gc_bif, '+', {f,0}, 0, [{x,0}, {x,1}], {x,2}}.
```

adds registers x[0] and x[1] together and puts the result into x[2], from which the DFG fragment in Fig. 8(a) are generated. Note that 32-bit datapath is assumed in this paper. In the case of addition, x[0]+x[1]−b'1111 results in addition of the upper 28 bits and setting of tag b'1111 in the lower 4 bits.

(2) List and tuple manipulation

Manipulation on list and tuple data are translated into a sequence of loads and stores on the heap. For example,

```
{get_list, {x,0}, {x,1}, {x,2}}.
```

takes list x[0], whose upper 30 bits represents the address of the first element and lower 2 bits are tag b'01, and extracts its first element (car) and remaining part (cdr) into x[1] and x[2], respectively. This is compiled into the DFG fragment in Fig. 8(b) which loads data from the heap.

(3) Jump and call

Jump instructions are translated into transition among DFGs. For example,

```
{test, is_noempty_list, {f,4}, {x,0}}.
```

verifies that the list pointed by x[0] is non-empty; if the test fails, the control is transferred to the instruction labeled by `f4`. It is translated to the conditional control transfer between DFGs, as shown in Fig. 8(c). A call instruction

```
{call, 1, {f,2}}.
```

saves the return address in CP, the continuation pointer, and jumps to `f2`. It is translated to an operation sequence to save the ID of the return instruction and to transfer the control. Return to the calling point is achieved based on the table which maps the instruction IDs to the states.

(4) Manipulation of the heap and the stack

When the instructions to secure memory cells on the heap or the stack do not find enough free cells, they trigger garbage collection (GC), which are processed by the library module attached to the process module. Thus, these instructions are converted into a DFG fragment to call the library module which consists of 1) stores of arguments, 2) store to variable to activate the library module, 3) polling to wait for the completion of the library module, and 4) loads of the results.

(5) Message passing

Message passing also involves complex tasks such as copy of heap data and garbage collection, which are also converted into a DFG fragment to call the library module.

C. *Port Module*

It is assumed that a byte sequence arriving at an input port is kept in a buffer attached to the port and that an outgoing byte sequence from an output port is written into a buffer attached to the output port. For each output port, a library module is created. On receiving messages from processes, the library module for the output port interprets the message data in the internal format and writes the corresponding byte sequences into the buffer. Input ports do not have dedicated hardware; message transfer is handled by the library modules of the receiving processes (as is described in the next subsection).

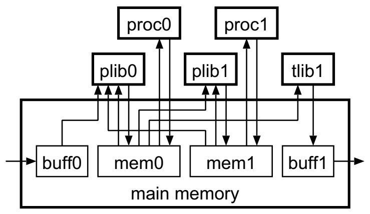


Fig. 9. Hardware configuration for the model in Fig. 5(a).

D. *Library Module*

A Library module is in charge of complex tasks such as heap manipulation and message passing.

In the method presented in this paper, one library module is provided for a process or an output port. The library module reads and writes the local memory (the heap and the stack) for the process, or the buffer of the output port. It also reads the local memories for the processes or the buffers for the input ports which may send messages to the process. For example, in the case of the example in Fig. 5(a), the configuration of the hardware is as shown in Fig. 9. Process modules proc0 and proc1 read/write their own local memories mem0 and mem1, respectively. Incoming and outgoing byte sequences are stored in buffer0 and buffer1, respectively. plib0 and plib1 are the library modules of proc0 and proc1, respectively, and tlib1 is the library module for port1. plib0 reads/writes mem0 as well as it references mem1 and port0, for there may be messages from proc0 and the input port. On receiving message from proc0 to port1, tlib1 reads mem0 and writes decoded data to buff1.

The library module executes the following six functions.

(1) test_heap *m*, *n*

Test if *m* free words are available on the heap. If not, call garbage collection. *n* is the number of the x registers which must be protected from garbage collection.

(2) allocate *m*, *n*

Expand the stack region by *m*+1 words by updating SP. If necessary free words are not available in the local memory, do garbage collection.

(3) send

Send a value in x[1] as a message to the process or the port indicated by x[0].

In the method of this paper, queueing of the message and copy of its accompanying heap data are handled by the library module of the recipient. So, this task just sets the flag to notify the existence of the new message. The flag is prepared for each of all the possible combinations of sender processes/ports and receiver processes/ports.

The library module of the recipient process polls the flag, and as soon as the flag is set, it enqueues the message and copies the accompanying data to the mini heap. If a process receives messages from multiple senders, the messages are processed one by one. After the queueing process, the library module of the receiver resets the flag, then the sender library module returns the control to the process module.

(4) receive

Copy the current message in the queue to x[0] and copy any data attached to the message. If enough free words are not available in the heap, do garbage collection.

(5) remove_message

Deletes the current message, which matches with a certain pattern, from the message queue.

(6) save_message

Just proceed the current message pointer by one when all the matches on the current message fail.

The library modules are controlled by means of polling. Let RUN_i be the variable or the register to control the library module of process i. When RUN_i is 0, the library module is idle. When the process want to activate the library module, it writes the number (1 through 6) of the desired function into RUN_i after writing the argument values into the local memory. Then the library module executes the function indicated by RUN_i, stores the result in the memory, and resets RUN_i. As soon as RUN_i is turned off, the process module loads the result and resumes its tasks.

V. Implementation

A prototype high-level synthesizer based on the proposed method has been implemented which runs on Ubuntu Linux and Mac OS X. The BEAM to CDFG translator (Ⓐ in Fig. 7) is implemented in Perl5. All the operations in CDFGs are based on 32-bit datapath of ACAP. The C library programs (Ⓑ in Fig. 7) are obtained by extracting and reducing the necessary portions from the source code of the BEAM interpreter of Erlang OTP 18.1.3. The original codes for handling the message queues and copy of the heap data were used almost without modification, though unnecessary codes are deleted and dynamic memory allocation was rewritten into static memory allocation. While the original version of the garbage collector in the BEAM interpreter is based on the mark-and-sweep method with two dynamic regions, our prototype adopted rather simple method which alternatively use two statically allocated regions. As for the data structure to bookkeep processes and the routines for message passing, simple versions that met our requirements were newly coded. The C programs were tested on a PC (with x86 CPU) and then synthesized into Verilog HDL codes by ACAP.

TABLE I summarizes the metrics of the FPGA based hardware synthesized from the Erlang specification in Fig. 5 (with the structure in Fig. 9). "LUTs", "FFs" and "delay" are the numbers of the LUTs and FFs, and the critical path delay, respectively, obtained by Xilinx ISE 14.3 targeting Spartan6. The size of the process modules is roughly proportional to the size of the BEAM assembly codes. Considering the amount of the tasks performed by the processes, the hardware may be too large. The size of the library modules is independent of the size of the processes. The current hardware is also considered to be a little too large. The reduction of the hardware size is definitely one of the most important next tasks in this project.

TABLE I
Synthesis result of Erlang code in Fig. 5.

	LUTs	FFs	delay [ns]
proc0	3,835	686	18.929
proc1	4,380	451	18.929
plib0	24,643	1435	21.414
plib1	23,916	1384	21.468
tlib1	24,405	1403	21.541

Logic synthesis: Xilinx ISE 14.3, target: Spartan6

VI. Conclusion

This paper has presented a method of high-level synthesis from control specification of embedded systems by Erlang. A prototype synthesizer has implemented which has generated Verilog codes from a simple example with communicating two processes.

Currently, the resulting hardware is too large for practical use. There are still many measures to reduce the hardware size, on which we are now working. Another drawback of the current method is that all the process modules access a single shared memory. Since this apparently produces bottlenecks, we are also working on a distributed memory architecture for the synthesized hardware.

Acknowledgement

Authors would like to express their appreciation to Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University. This work is partly supported by JSPS KAKENHI Grant Number 16K00088.

References

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: "High-Level Synthesis: Introduction to Chip and System Design," Kluwer Academic Publishers (1992).

[2] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced SystemBuilder: A tool set for multiprocessor design space exploration," in Proc. ISOCC 2010, pp. 79-82 (Nov. 2010).

[3] Joe Armstrong: *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf (2007).

[4] Brian Chamberlain: Using Erlang on the RaspberryPi to interact with the physical world (online), `http://www.slideshare.net/breakpointer/using-erlang-on-the-raspberrypi` (accessed 2016-02-04).

[5] N. Ishiura, H. Kanbara, and H. Tomiyama: "ACAP: Binary Synthesizer Based on MIPS Object Codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).