# Distributed Memory Architecture for High-Level Synthesis of Embedded Controllers from Erlang

Kagumi Azuma
Nagisa Ishiura
Department of Informatics
Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, Japan

Nobuaki Yoshida
Hiroyuki Kanbara
ASTEM RI/KYOTO
Kyoto, Japan

## Abstract

This paper presents a distributed memory architecture for dedicated hardware automatically synthesized from Erlang programs. Our team had developed a framework for generating embedded systems controllers whose behavior was specified by a subset of Erlang, where each process was mapped into hardware (a logic circuit) running independently of the circuits for the other processes. However, the resulting hardware was not of practical use because it shared a single main memory potentially accessed by all the circuits for the processes simultaneously. To address this issue, in this paper, the main memory is partitioned into banks so that each process can access its own memory independently of the other processes. In order to keep the interconnections for message passing to a practical size, a bus architecture is employed where send requests are arbitrated by an arbiter (logic circuit). In order to make the resulting hardware as small as possible, a garbage collection circuit is shared among the circuits for the processes also under the control of the arbiter. From a simple Erlang specification, Verilog HDL codes for necessary hardware to construct a system has been generated.

***CCS Concepts*** • **Computer systems organization → Embedded hardware**; • **Hardware → Hardware-software codesign**; • **Software and its engineering → Functional languages**;

***Keywords*** Erlang, high-level synthesis, distributed memory architecture, embedded systems, ACAP

**ACM Reference Format:**
Kagumi Azuma, Nagisa Ishiura, Nobuaki Yoshida, and Hiroyuki Kanbara. 2017. Distributed Memory Architecture for High-Level Synthesis of Embedded Controllers from Erlang. In *Proceedings of 16th ACM SIGPLAN International Workshop on Erlang, Oxford, UK, September 8, 2017 (Erlang'17),* 7 pages.
https://doi.org/10.1145/3123569.3123574

## 1 Introduction

Nowadays, innumerable embedded systems are implemented in various products, such as consumer electronics, automobiles, medical appliances, industrial electronics, autonomous robots, etc. In order to meet various needs for these products, higher and higher functionalities are being required.

Especially, 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. 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. It is a next challenge to establish new methodologies to model and to automate design of such communication oriented and sophisticated systems.

One approach to address this issue is to use domain specific languages which orient event processing, concurrency specification, and networking. Erlang [1] is a concurrency oriented functional programming language which is originally developed for implementing telephony switches. While it is widely used in the area of telecommunications, e-commerce, instant messaging, etc., there are some attempts to use Erlang for embedded systems [2], based on an observation that concurrent processes and message passing will allow succinct description of event processing.

On the other hand, at the same time with higher functionality, higher performance as well as lower power consumption is 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.

High-level synthesis [3], which automatically synthesizes hardware design from software programs, is one of the key technologies to expedite such re-implementation. Various methodologies to automate hardware design based on high-level synthesis have been proposed [4, 5]. If hardware is automatically derived from Erlang specification, advanced sophisticated systems would be easily implemented as efficient devices with higher response and smaller power consumption than processor-based systems.

In this context, Takebayashi et al. [6] proposed a way of specifying the behavior of embedded systems in a subset of Erlang and a method of synthesizing hardware description from the specification. In their framework, each Erlang process is implemented as a separate logic circuit. Assembly codes of BEAM virtual machine obtained by compiling the input Erlang programs are translated into CDFG (control dataflow graph, a typical intermediate data structure for high-level synthesis), from which Verilog HDL [7] codes are generated using the back-end of the high-level synthesizer ACAP [5]. Although, they generated Verilog HDL codes from a simple Erlang specification, they were not of practical use because it assumed a shared centralized memory; it is accessed by all the circuits for the processes simultaneously but a memory device with such


*Erlang'17, September 8, 2017, Oxford, UK*

ACM ISBN 978-1-4503-5179-9/17/09…$15.00
https://doi.org/10.1145/3123569.3123574

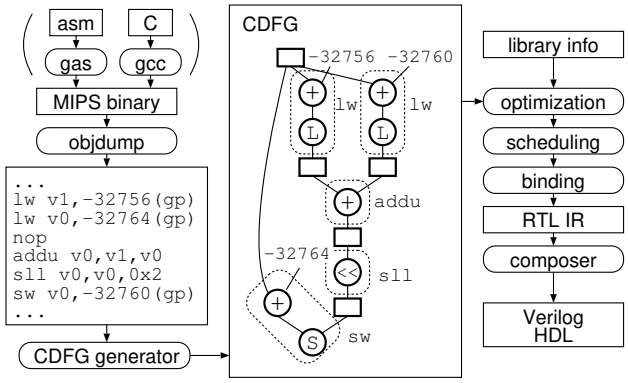



**Figure 1.** High-level synthesizer flow in ACAP [5].

many ports does not exist or is not affordable at least to embedded systems.

To address this issue, this paper presents a distributed memory architecture for high-level synthesis of embedded systems from Erlang. In this architecture, the stack, the heap, and the message queue of each process are stored in its local memory, which can be accessed independently of the other processes. The local memories are connected with two buses, through which messages are passed around. Send operations using the bus is arbitrated by an arbiter circuit. We have implemented a prototype of the synthesizer on the bases of ACAP, and from a simple Erlang specification, Verilog HDL codes for necessary hardware to construct a system has been generated.

## 2 High-Level Synthesis and ACAP

High-level synthesis [3] is a technology to generate dedicated hardware automatically from a behavioral specification of target systems, which is often described in procedural programming languages such as C. The typical flow of synthesis is as follows:

1. CDFG generation
   A given specification is compiled into an intermediate data structure called a control dataflow graph (CDFG), which is a collection of dataflow graphs (DFGs) and transition relations among them. A dataflow graph expresses the operations and dependencies among them in a basic block.
2. Scheduling
   The clock cycle, or the time frame, to execute each operation in each DFG is determined. Typically, this is a problem of minimizing total clock cycles to execute all the operations in each DFG with the given numbers of execution units.
3. Binding
   Each operation is assigned to a concrete instance of the execution unit and each value is assigned to a register (a hardware unit to memorize a value, consisting of a set of flip-flops). This determines the topology of the resulting hardware, so it is formulated as the problem of minimizing the estimated size of the hardware.
4. HDL generation
   The specification of the hardware is generated in the form of a hardware design language (HDL), such as Verilog HDL [7] or VHDL [8].

ACAP [5] is one of the high-level synthesis system, which we utilize as a base of the work in this paper. It synthesizes Verilog HDL from C programs or MIPS binary codes. The flow of synthesis is shown in Figure 1. 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. Verilog HDL is generated according to the standard high-level synthesis procedure.

We use ACAP as a basis of the work in this paper in two ways; we utilize the back-end of ACAP (scheduling through HDL generation) to generate Verilog HDL from Erlang via CDFG representation, and to obtain hardware modules for complex tasks such as message passing and garbage collection by high-level synthesis from C codes in the BEAM emulator. Any other high-level synthesizer than ACAP may be used as long as 1) it accepts CDFGs as well as C codes as inputs, and 2) it can compile ANSI-C programs into HDL codes.

## 3 High-Level Synthesis from Erlang

### 3.1 Erlang Subset for Specifying Embedded Systems Control

The systems we deal with in our current project [6] are relatively small ones that control embedded systems. It consists of fixed number of processes up to 10. Furthermore, considering hardware implementation, we assume a very limited subset of Erlang in this paper. Since the number of the processes are fixed, it is assumed that all the processes are created at the 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. Currently, communication only within the system is handled. Namely, communication with external processes via TCP/IP is out of the scope of this paper. The data types handled in this paper are limited to 28 bit signed integers, lists, and tuples. Closures are not supported.

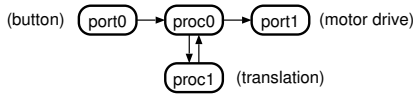
Figure 2 shows a small example of control description by our Erlang subset. The controller receives signals at an input port `port0` from button presses indicating the directions to move. It sends out corresponding signals to the driving device at an output port `port1`. A process `proc0` handles I/O and a process `proc1` is in charge of data translation. In the Erlang code, the two ports and two processes are created in `start` (lines 4–14).

### 3.2 Flow of Synthesis

In the area of hardware design, a circuit to perform a certain task is called a (hardware) module, so we will henceforth refer to the circuit for a process as a process module. Similarly, the circuits to perform arbitration, bus functions, and garbage collection will be called an arbiter module, a bus module, and a garbage collection module, respectively.

In our 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.

An Erlang process may execute multiple functions. In our method, all the functions executed by each process, which are recognized by static analysis, are synthesized into a single hardware module. For example, a hardware module for process `proc0` in Figure 2 (b) should be able to execute functions `loop0` and `decode`, while the `proc1` module should run functions `loop1`, `calc`, and `encode`.




(a) Communication among processes and ports.

```
01: -module(roomba).
02: -export(start/0).
03:
04: start() ->
05:   spawn(fun() ->
06:     register(proc1, self()),
07:     loop1(0, 0)
08:   end),
09:   spawn(fun() ->
10:     register(proc0, self()),
11:     Port0 = open_port({spawn, "stdbuf -i0 -o0 -e0 od
   -h -w8 /dev/input/js0 | ./controller"}, {packet, 2}),
12:     Port1 = open_port({spawn, "./roomba"}, {packet, 2}),
13:     loop0(Port0, Port1)
14:   end).
15:
16: decode(Dt,Dh,Et,Eh) ->
      {((Dh bsl 8) bor Dt), ((Eh bsl 8) bor Et)};
17: decode(X) -> X.
18:
19: loop0(Port0, Port1) ->
20:   receive
21:     {Port0, {data, Data}} ->
22:       Data2 = decode(Data),
23:       proc1 ! {proc0, data, Data2},
24:       loop0(Port0, Port1);
25:     {proc1, Data3} ->
26:       Port1 ! {proc0, {command, Data3}},
27:       loop0(Port0, Port1);
28:     {Port1, _} ->
29:       loop0(Port0, Port1);
30:     _ ->
31:       loop0(Port0, Port1)
32:   end.
33:
34: loop1(D, T) ->
35:   receive
36:     {proc0, data, Data} ->
37:       {Drive, Turn} = calc(Data, D, T),
38:       Cmd = encode(Drive, Turn),
39:       proc0 ! {proc1, Cmd},
40:       loop1(Drive, Turn);
41:     X ->
42:       proc0 ! X,
43:       loop1(D, T)
44:   end.
45:
46: calc({Para, X}, Drive, Turn) ->
47:   if
48:     X == 258  -> {Para, Turn};
49:     X == 1026 -> {Para, Turn};
50:     X == 2    -> {Drive, Para};
51:     X == 770  -> {Drive, Para};
52:     true      -> {0, 0}
53:   end.
54:
55: encode(Drive, Turn) ->
56:   if
57:     Drive =< 57343, Drive >= 32768 ->
58:       if
59:         Turn =< 57343, Turn >= 32768 -> {146, 0, 127, 0, 63};
60:         Turn =< 32767, Turn >= 12288 -> {146, 0, 63, 0, 127};
61:         true                         -> {146, 0, 127, 0, 127}
62:       end;
63:     Drive =< 32767, Drive >= 8192 ->
64:       if
65:         Turn =< 57343, Turn >= 32768 -> {146,255,127,255,63};
66:         Turn =< 32767, Turn >= 12288 -> {146,255,63,255,127};
67:         true                         -> {146,255,127,255,127}
68:       end;
69:     true ->
70:       if
71:         Turn =< 57343, Turn >= 32768 -> {146,255,127,0,127};
72:         Turn =< 32767, Turn >= 12288 -> {146,0,127,255,127};
73:         true                         -> {146,0,0,0,0}
74:       end
75:   end.
```

(b) Behavior description by Erlang [6].

**Figure 2.** Example of Erlang description.

When multiple processes call the same function, the code for the function is duplicated.

The flow of synthesis is illustrated in Figure 3. 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 generated. However, some BEAM instructions involve complex tasks such as message passing and garbage collection

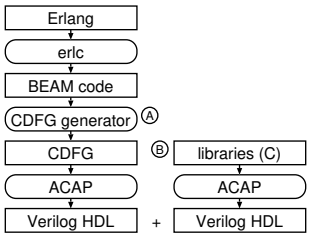
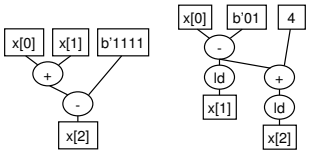



**Figure 3.** Flow of hardware synthesis from Erlang via BEAM code [6].




(a) addition (b) get_list

**Figure 4.** Conversion from BEAM instruction to DFG [6].

which would be difficult to embed into the CDFG. In our framework, these tasks are implemented as separate hardware modules, called "library modules," which are activated from the CDFG per demand. The library modules are synthesized from a reduced C code of the BEAM interpreter by ACAP.

### 3.3 Converting BEAM Assembly to CDFG

The BEAM assembly code 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 are converted into operations of a DFG (dataflow graph), and then 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 in DFGs. Since the 28 bit integer data 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, for an instruction
   ```
   {gc_bif,'+',{f,0},0,[{x,0},{x,1}],{x,2}}.
   ```
   which adds the contents of the registers x[0] and x[1] of the BEAM VM together and puts the result into x[2], the DFG fragment in Figure 4 (a) is 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 of lists and tuples is translated into a sequence of loads and stores on the heap. For example, for an instruction
   ```
   {get_list,{x,0},{x,1},{x,2}}.
   ```

which 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, is compiled into the DFG fragment in Figure 4 (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 DFG corresponding to the label `f4`. 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. Returning 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.

### 3.4 Library Module

Each process is provided with a library module which handles complex tasks. The library module executes the following seven functions.

1. test_heap *m*, *n*
   Tests if *m* free words are available on the heap for the process. If not, garbage collection is invoked. *n* is the number of the x registers which must be protected from garbage collection.
2. allocate *m*, *n*
   Expands the stack region by *m*+1 words by updating SP. Lack of necessary free words triggers garbage collection.
3. send
   Sends the contents of x[1] register as a message to the process or the port indicated by x[0] register. If the destination is a process, it enqueues the message to the message queue of the destination process and copies the accompanying data, if any, to the "mini heap" attached to the message. If the message is destined to an output port, the message and the accompanying data are serialized and put into the byte stream buffer of the output port. The send function may be invoked by an input port when a new byte sequence is detected on the input buffer of the port. In this case, a list data structure is assembled which are copied to the message queue and the mini heap of the destination process.
4. receive
   Copies the current message in the queue to x[0] register and copy any attached data into the heap. If enough free words are not available in the heap, garbage collection is invoked.
5. remove_message
   Unlinks the current message from the message queue (when a match succeeded on the message).
6. save_message
   Proceeds the current message pointer by one (when all the matches on the message fail).
7. wait_timeout *t*
   Waits for a new message. If no message arrives within *t* clock cycles, quit waiting (and just returns the control to the caller).

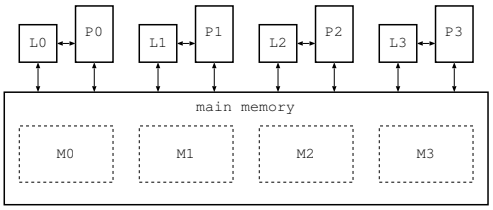



**Figure 5.** Previous hardware configuration with single main memory.

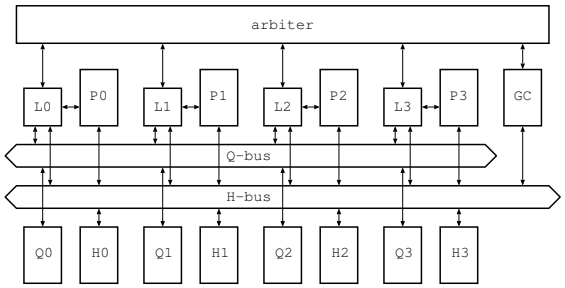



**Figure 6.** Proposed hardware configuration with distributed memories.

The library modules are controlled by process modules via control variables. Let RUN_*i* be the variable 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 7) of the desired function into RUN_*i* after writing the argument values into the memory. Then the library module completes the execution of the function indicated by RUN_*i*, it 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.

## 4 Distributed Memory Architecture

In the synthesis method proposed in [6], a centralized memory architecture is assumed, as shown in Figure 5, where a single memory module is accessed by process modules and library modules in parallel. This is obviously unrealistic, for a memory module of as much as 10 ports is not affordable at least to embedded systems. The other problem with the previous architecture is that the size of the hardware, especially that of library modules, is too large. This

is because, each library module must include hardware for garbage collection which is large but not used so frequently.

In order to make the synthesized hardware more practical, we propose in this paper a distributed memory architecture, illustrated in Figure 6, which has the following features:

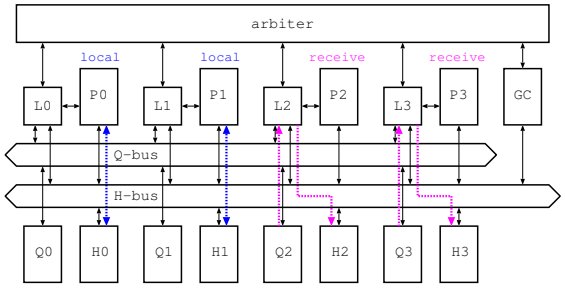
1. Distributed memories
   Each process and library module (P*i* and L*i*) have their own memory, which is further partitioned into a memory for the heap and the stack (H*i*) and the one for the message queue and the mini heap (Q*i*). Note that only a single port is enough for both H*i* and Q*i*. Nevertheless, all the processes can operate independently with each other, except for during message passing and garbage collection.
2. Shared garbage collector
   A single garbage collection module (GC) is shared with all the processes. This contribute to significant reduction on the hardware cost, though only one process can execute GC at the same time.
3. Two buses for global communication
   Two buses, Q-bus for accessing Q*i*s and H-bus for H*i*s are provided to carry out message passing and garbage collection. We adopt bus architecture for message passing because a point-to-point network needs huge amount of interconnects and support hardware. However, with the bus architecture, message passing and GC must be processed one by one.
4. Arbitration
   Message passing and GC are moderated by an arbiter module. It receives requests for GC or message passing and permits their execution according to predefined priorities.

### 4.1 Distributed Memory and Bus Modules

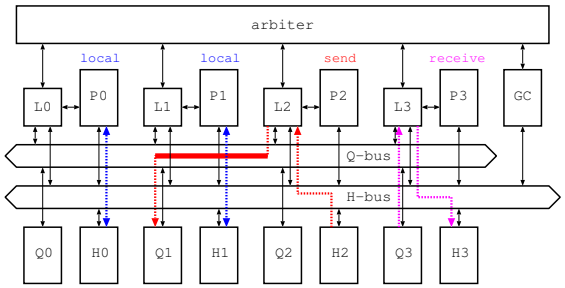
The distributed memory scheme works as follows.

1. Each process module P*i* can access its own working memory H*i* independently of the other processes. For example, as shown in Figure 7 (a), P0 and P1 can execute their local computation using their working memories in parallel. Each library module L*i* may access message queue Q*i* and H*i* independently of the other processes. So receive operations can be also executed in parallel.
2. When process P*i* sends a message to P*j*, it reads its own working memory H*i* and writes to the message queue Q*j* of the destination process. The latter is done through the Q-bus. In Figure 7 (b), for example, P2 is sending a message to P1, so L2 reads H2 and writes to Q1. Meanwhile, irrelevant P0 and P3 can operate in parallel. Furthermore, P1 can execute local computation, as long as it does not access Q1. (In this case, P1 cannot execute receive.)
3. When process P*i* requests GC, the GC module cleans up H*i* via H-bus. In Figure 7 (c), garbage collection is being executed on the heap of P1. The other processes can run in parallel as long as they do not access H1. Thus, even the send operation from P2 to P1 can be done simultaneously.

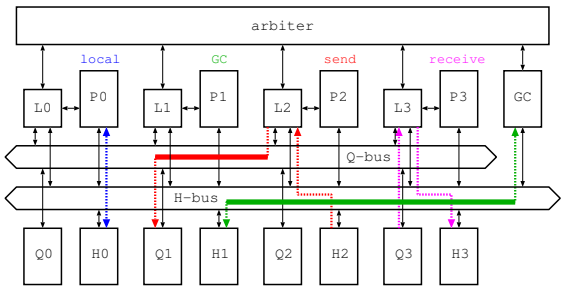
Accesses to the memories of other processes are handled by a bus module. We assume a single address space for an entire system, which is addressed by 32 bits. Each of H*i* or Q*i* is assigned a segment from the address space. The lower $m$ bits of the address are used for the local memory address and the upper $32 - m$ bits are used for distinguishing the segments.




(a) Parallel execution of local operations.




(b) Send operation.




(c) Garbage collection.

**Figure 7.** Distributed memory architecture.

On the side of L*i*, read accesses to the Q*i* and Q*j* ($j \neq i$) are handled as follows:

- If the read access is from L*i* and the segment number (the upper $32 - m$ bits) of the address is $i$, then the read request is forwarded to Q*i* with the lower $m$ bit of the address, and the answer from Q*i* is returned to L*i*.
- If the read access is from L*i* and the segment number of the address is not $i$, then the read request is broadcast via Q-bus with the full address. As soon as the resulting data appear on the bus, they are returned to L*i*.
- If the read access is from the Q-bus (namely, from the other processes) and the segment number of the address is equal to $i$, then the read request is forwarded to Q*i* with the lower $m$ bit of the address, and the answer from Q*i* is put on the bus.

Note that no two library modules try to access the Q-bus simultaneously. Furthermore, no two L*i* and L*j* do not access Q*i* simultaneously. These are guaranteed by the arbiter.

The accesses to H*i* are handled in the same way as those to Q*i*, except that both L*i* and P*i* may access H*i*. This, however, does not

make big difference because L$i$ and P$i$ does not access H$i$ simultaneously.

### 4.2 Arbiter

The arbiter module arbitrates message passing and garbage collection based on the requests from the processes. If a request arrives while the previous one is being processed, the new one will be processed after the previous one. If multiple requests are waiting or multiple requests arrive simultaneously, the order is determined according to the predetermined priority.

Requests for garbage collection are processed as follows.

1. Library module L$i$ requests GC by raising its GC request signal GC_req$_i$ to 1.
2. On noticing GC_req$_i$=1, the arbiter forwards the request to the GC module by setting GC_req=1 and GC_process=$i$. If multiple requests are arriving, the arbiter chooses one of them according to the priority.
3. The GC module does garbage collection on Q$i$, and then notifies the completion by making GC_req=0.
4. The arbiter forwards the notification to L$i$ by setting GC_req$_i$=0.
5. On confirming GC_req$_i$=0, L$i$ and P$i$ resume their execution.

The protocol for send requests needs extra steps because the send is not permitted while the destination process is executing a receive operation (which manipulates the message queue).

1. Library module L$i$ requests send to process $j$ by setting send_req$_i$=1 and send_to=$j$.
2. Arbiter forwards the request to process $j$ by raising enqueue request signal of $j$ (enq_req$_j$=1). If multiple requests are arriving, the arbiter chooses one of them according to the priority.
3. If the process $j$ is not executing neither of receive_message, remove_message, nor save_message, then it permits enqueueing of the new message by setting enq_ready$_j$=1.
4. The arbiter forwards the permission to L$i$ by setting send_req$_i$=1.
5. On confirming send_req$_i$=1, L$i$ starts enqueueing of the message and copy of the heap data to Q$j$.
6. L$i$ notifies the completion of the send operation by setting send_req$_i$=0.
7. Arbiter forwards the completion by setting enq_req$j$=0.

Note that library module L$j$ must be designed carefully so that it can respond to enq_req$_j$ = 1 while it is waiting for its send request (of the lower priority) to be processed.

### 4.3 I/O Modules

In order to access to the signals regarding arbitration, such as GC_req$_i$ and enq_ready$_j$, in a uniform manner, we place them all in the address space. Namely, all the signals are read and written to by memory mapped I/O. This interfacing is taken care of by I/O modules. Figure 8 is a revised version of Figure 6.

## 5 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 Figure 3) is implemented in Perl5. All the operations in CDFGs are based on 32 bit datapath of ACAP.

The C library programs (Ⓑ in Figure 3) are obtained by extracting and reducing the necessary portions from the source code of the

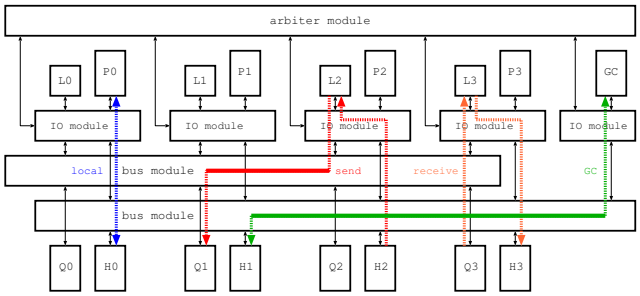



**Figure 8.** Proposed architecture with I/O modules.

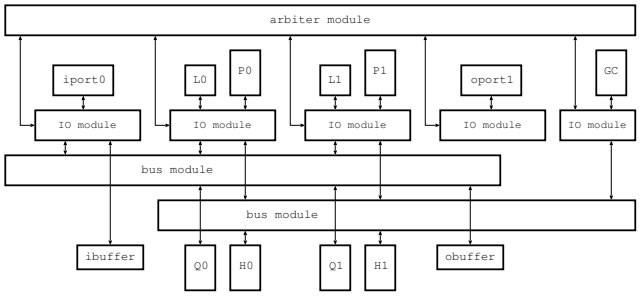



**Figure 9.** Synthesized hardware configuration of the sample program.

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. Verilog HDL codes of the library modules L0, L1, iport0 and oport1 were synthesized from these C programs by ACAP.

TABLE 1 summarizes the metrics of the hardware, implemented as circuits in an FPGA (field programmable gate array; a programmable logic device), synthesized from the Erlang specification in Figure 2 (with the structure in Figure 9). "LUTs", "FFs" and "delay" are the numbers of the LUTs (look-up tables, or the logic gates of the FPGA) and flip-flops (registers), and the critical path delay, respectively, obtained by Xilinx Vivado (2015.4) targeting Artix-7 (xc7a100tcsg324-3). For the readers' reference, the figures for a MIPS R3000 compatible CPU core are listed.

The bus, arbiter, and I/O modules are small enough, for they have been manually designed in Verilog HDL. Considering the amount of the tasks performed by the processes, the hardware may be too large. The sizes of the library modules are also a little too large. This is because we have just succeeded in generating Verilog HDL from Erlang or C codes and have not done much optimization yet.

We have not tested the behavior of the synthesized Verilog codes, though the behavior of the library modules were tested in the C program level on PC (with x86 CPU), and the protocols regarding

**Table 1.** Synthesis result of Erlang code in Figure 2.

| | LUTs | FFs | delay [ns] |
|---|---|---|---|
| bus | 172 | 212 | 3.654 |
| arbiter | 156 | 51 | 3.659 |
| I/O | 300 | 162 | 5.825 |
| P0 | 4,480 | 806 | 8.836 |
| P1 | 4,788 | 741 | 9.217 |
| GC | 13,185 | 1,260 | 12.828 |
| L0 | 12,338 | 1,346 | 13.257 |
| L1 | 12,023 | 1,380 | 13.447 |
| iport0 | 10,700 | 1,259 | 12.652 |
| oport1 | 496 | 162 | 5.625 |
| MIPS R3000 | 3,166 | 1,773 | 11.698 |

Logic synthesis: Vivado 2015.4, target: Artix-7

arbitration of send and GC requests were tested by cycle-accurate simulation with models written in C.

## 6 Conclusion

This paper has presented a distributed memory architecture for high-level synthesis from control specification of embedded systems by Erlang. A prototype synthesizer has been implemented which has generated Verilog codes from a simple example.

Currently, the resulting hardware is still too large for practical use. We are now working on reduction of redundancies and various optimization measures on process modules and library modules. For example, a library modules is almost a finite state machine, except for functions to copy heap data structs and message queue manipulation, so it can be written in Verilog HDL rather than C, which drastically reduces the resulting hardware size. There are also much room for optimization in generating CDFG from BEAM codes.

## Acknowledgments

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 like to thank Mr. Hinata Takebayashi (formerly with Kwansei Gakuin University) who have developed the initial version of the BEAM to CDFG compiler. We would also like to thank to all the members of Ishiura Lab. of Kwansei Gakuin University. This work is partly supported by JSPS KAKENHI under Grant Nos. 16K00088 and 16K01207.

## References

[1] Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf.
[2] Brian Chamberlain. Using Erlang on the RaspberryPi to Interact with the Physical World. Retrieved February 4, 2016 from http://www.slideshare.net/breakpointer/using-erlang-on-the-raspberrypi.
[3] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. 1992. High-Level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers.
[4] Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, and Hiroaki Takada. 2010. Advanced SystemBuilder: A Tool Set for Multiprocessor Design Space Exploration. In Proceedings of the International SoC Design Conference (ISOCC 2010). (November 2010). 79–82.
[5] Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama. ACAP: Binary Synthesizer Based on MIPS Object Codes. In *Proceedings of International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2014)*. (July 2014). 725–728.
[6] Hinata Takebayashi, Nagisa Ishiura, Kagumi Azuma, Nobuaki Yoshida, and Hiroyuki Kanbara. 2016. High-Level Synthesis of Embedded Systems Controller from Erlang. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*. (October 2016). 285–290.
[7] IEEE Computer Society. 2001. IEEE Standard Verilog Description Language (IEEE Standard 1364-2001), IEEE, New York, NY, USA.
[8] IEEE Computer Society. 2008. IEEE Standard VHDL Language (IEEE Standard 1076-2008), IEEE, New York, NY, USA.