ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer

Masahiro Yasugi Satoshi Matsuoka
Akinori Yonezawa

Department of Information Science, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan*

Abstract

The trend towards object-oriented software construction is becoming more and more prevalent, and parallel programming cannot be an exception. In the context of parallel computation, it is often natural to model the computation as message passing between autonomous, concurrently active objects. The problem was, as some previous studies had indicated, that the overhead from message reception to dynamic method dispatching consumes a significant amount of execution time (e.g., as much as 4000 machine cycles or 500 μseconds at 8 MHz clock for some language/hardware combination). Our **ABCL/onEM-4**, a software/hardware implementation architecture for a concurrent object-oriented language, overcomes this problem with technologies such as address-specifiable reactive packet-driven architecture, zero-overhead context switching, and packet-driven allocation of message boxes. Preliminary performance measurements on a real hardware EM-4 confirm our claim, achieving the performance of up to nearly 10 μseconds (130 clocks) total for a remote object-creation followed by a request message send to the created object and a reply reception from the object, for a 12.5 MHz clock speed. Our results indicate that the concurrent object-oriented computational model and languages are highly viable with proper implementational software/hardware architectures.

1 Introduction

The trend towards object-oriented (OO) software construction is becoming more and more prevalent. Important software concepts such as *encapsulation* promote a high degree of code re-use and clean architectural structuring of large software. High-performance parallel programming, previously performed in the context of more conventional programming languages, would also be able to enjoy the benefit from the OO technology with appropriate OO languages and systems.

*E-mail: {yasugi,matsu,yonezawa}@is.s.u-tokyo.ac.jp

Although many OO languages currently in use today (such as C++ and Smalltalk) are sequential, it is more natural to consider objects as being a unit of concurrency in the context of parallel computation. There, the computation is modeled as message passing between autonomous, concurrently active objects. A recent breed of OOCP (*Object-Oriented Concurrent Programming*) languages attempt to provide maximum computational and modeling power through concurrency of objects. Recent work has also been successful in establishing strong, theoretical foundations for concurrent objects[1, 15, 2, 12, 6].

The implementation of efficient OOCP languages, unfortunately, had not been as successful. The problem was that the overhead from message reception to dynamic method dispatching consumes a significant amount of execution time (e.g., as much as 4000 machine cycles or 500 μseconds at 8 MHz clock rate for A-NET[4], a OO-concurrent software/hardware architecture). The indications were that, even with the support of hardware, the cost of message passing could be significant.

Our proposed software/hardware architecture for a concurrent object-oriented language (called **ABCL/on EM-4**) overcomes the problems with a combination of software/hardware technologies such as *address-specifiable reactive packet-driven architecture, zero-overhead context switching, and packet-driven allocation of message boxes*. Preliminary performance measurements on a real hardware (EM-4) confirm our claim, achieving the performance of up to nearly 10 μseconds (130 clocks) total for a remote object-creation followed by a request message send to the created object and a reply reception from the object, for the clock-speed of 12.5 MHz. Compared to our other implementation work of an OOCP language on a more conventional multicomputer without provisions for concurrent-OO style computing (Fujitsu AP1000[9], a 512 node multicomputer based on SPARC chips), we have been able to achieve an order of magnitude improvement in inter-node message passing latency (approximately 35 μseconds vs. a few μ seconds). Even compared to the Cosmos/J-Machine[5], which is highly optimized for concurrent-OO computation, our required machine cycles are considerably smaller. Our results indicate that the concurrent object-oriented computational model and languages are highly viable with a proper implementational software/hardware architecture combination.

The remainder of this paper is structured as follows: Section 2 briefly describes the concurrent OO computation model. Section 3 overviews the hardware architecture of EM-4. Section 4 covers the software architecture of ABCL/onEM-4, specifically on the software technology for achieving fast message passing. Section 5 presents the actual performance measurements, and in Section 6, we discuss why our hardware/software implementational architecture incurs less overhead for concurrent OO style of computation compared to the previous proposals, such as J-Machine and A-NET. We finally conclude in Section 7.

2 Overview of Our Computation/Programming Model

In our computation/programming model[19, 18], computation is performed by a collection of software modules called *concurrent objects* which become active when they accept messages (*message-driven* or *reactive*), and computation is carried out by message transmissions among concurrent objects. More than one message transmission may take place in parallel and more than one object may become active simultaneously.

The unit of concurrency is a *concurrent object*. Each has its own (autonomous) single thread of control, and it may have its own local persistent memory, the contents of which can be accessed only by itself[1] (*encapsulation*). The state of a concurrent object at a given time is characterized by the contents of its local memory and the *mode* of its execution at that time.

Upon accepting a message, a concurrent object can execute a sequence of four kinds of basic actions:

1. *Message sends* to other concurrent objects.

   Messages could be *past type* (namely, "asynchronously send and no-wait", syntactically denoted by `[`*TargetObj* `<=` *Msg*`]`) or *now type* (namely, "asynchronously send and wait for a reply", syntactically denoted by `[`*TargetObj* `<==` *Msg*`]`).

2. *Creation* of concurrent objects (*dynamic creation*).

3. *Referencing and Updating* of the contents of its local memory (*imperative computation*).

4. *Various operations* (such as arithmetic operations) on values that are stored in its local memory and passed around by message transmissions.

A message may contain names (IDs, or global addresses) of concurrent objects as well as message tags and basic values such as numbers and booleans. Whether or not a message is acceptable is determined by the *message pattern*, the values contained in the message, and the current state and mode of the concurrent object. Therefore, in defining the behavior of a concurrent object, one must specify (1) how its local persistent memory is represented, (2) on what condition which message is accepted, and (3) what sequence of actions to be performed when a message is accepted. In our programming language, the local persistent memory is represented by a collection of variables, called *state variables*, and the description/code of each sequence of actions is called a *method (or script)*. Determining which method is executed is called *method look-up (or script look-up)*.

[1] It is possible for a concurrent object to be *pure*, i.e., not to have such local memory if not needed.

As noted above, *past* and *now* type message transmissions are asynchronous in the sense that the sender can send a message to a receiver (both are, of course, concurrent objects) at any time it wishes to. The messages arriving at a concurrent object are buffered in a message queue associated with the concurrent object until they are processed (*asynchronous buffered communication*). When two messages are sent from the same sender to the same receiver, they arrive at the receiver in the same order as they were sent (*preservation of transmission order*). A sender can send a message to any other concurrent object as long as the sender knows its name. Since the names of concurrent objects can be passed around by message transmissions, connection topology of concurrent objects is dynamic.

In accepting a message, a concurrent object may be in one of two modes, *dormant* and *waiting*. In the dormant mode, a concurrent object checks all the messages in its message queue from the head and deletes the messages until it finds an acceptable message, and it becomes *active* and processes the acceptable message (namely, deleting the message and executing the corresponding method). In the waiting mode, a concurrent object also checks its message queue from it head and skips all messages (without deleting them) until it finds an acceptable message, and it processes the acceptable message. When a concurrent object is (1) created, or, (2) immediately after every execution of one of its method, a concurrent object becomes dormant. During execution of a method, a concurrent object changes its mode from active to waiting when it performs a special *wait-for* command or *now type* message passing.

Our family of languages, ABCL, is based on this computation model, and have been well studied from theoretical aspects to applicational ones[19, 18]. Applications include parallel discrete event simulation[16], symbolic computation[20], and numerical computation[17].

3 Architectural Features of EM-4

Our base hardware architecture is EM-4, which was designed and built by Electro-Technical Laboratory (ETL) in Tsukuba, Japan. This section summarizes EM-4's features essential to our software/hardware architecture for a concurrent-OO language. More details of EM-4 can be found in [13, 11].

EM-4 was originally designed to support the *strongly connected arc dataflow model*, which extends the dataflow model by introducing a new type of arcs called *strongly connected arcs* into dataflow graphs. A dataflow subgraph whose nodes are connected by strongly connected arcs is called the *strongly connected block (SCB)*. The nodes in a SCB are executed prior to the nodes not in the SCB. The dataflow synchronization/matching is only applied to the initial node in a SCB.

Architecturally, EM-4 is a combination of a reactive, packet-driven dataflow architecture and a conventional register-based control-flow architecture. A packet received by a PE (processing element) is first stored in a hardware queue buffer. The PE takes a single packet from the queue, and performs a simple but fast dataflow matching by the use of the *direct matching scheme (see* *below)*. If the match occurs, it triggers the execution of a SCB on the PE. The execution of SCB is performed with efficient register-based pipeline execution, and is non-interruptible by subsequent packets until the execution is completed. Imperative operations such as memory writes are also possible, although not explicitly exploited in the strongly connected arc dataflow model. From a different perspective, one could regard a PE in EM-4 as a conventional RISC augmented with packet-driven and packet-output features.

The instruction sequence corresponding to a SCB is stored in a segment of memory called a *template segment*. (A single template segment can contain multiple instruction sequences.) In order to execute the instruction sequence corresponding to a SCB, a segment of memory called the *operand segment* is allocated, and the address of the template segment containing the instruction sequence is stored in the first word of the operand segment.

As mentioned above, matching is performed with a new scheme called the *direct matching scheme*, which is the most characteristic feature of EM-4. we briefly introduce the structure of packets for its explanation. A (normal) packet contains the following data (Figure 1(a)):

1. an operand segment number (*OSN*)
2. a displacement (*DPL*) from the top of an operand segment,
3. an operand data for the first instruction of a SCB.

Suppose that the first instruction of a SCB requires two operand data. When the first packet containing an operand data arrives, the operand data is stored in the matching memory whose address is obtained by adding DPL to the address of the top of operand segment obtained from OSN. When another packet containing the second operand data with the same OSN and DPL arrives subsequently, the instruction address to begin the execution of SCB is obtained by adding the DPL value to the address of the top of the template segment, which is pointed to the top word of the operand segment. Then the execution of the SCB is triggered from that address, with the two data operands of the two packets received (Figure 1(a)). If the first instruction of the SCB requires only one operand data, the execution of the SCB starts upon the arrival of the first packet without any matching. In both cases, the address of the operand segment is set into a special register called *operand segment register*.

In addition to the normal type of packets described above, there are more than ten types of special packets. The purpose of special packets is to invoke SCBs which do not need dynamic linkage of operand segments to template segments. A special packet specifies a predefined SCB, used for common system tasks such as allocating memory blocks in a remote PE (Figure 1(b)). In addition, a special packet contains two data items, an address data and an operand data, whereas a normal packet can only contain a single operand data.

The instruction set of EM-4 consists of arithmetic and logical instructions, test and branch instructions, data transfer instructions and communication instructions. Arithmetic, logical and communication instructions can

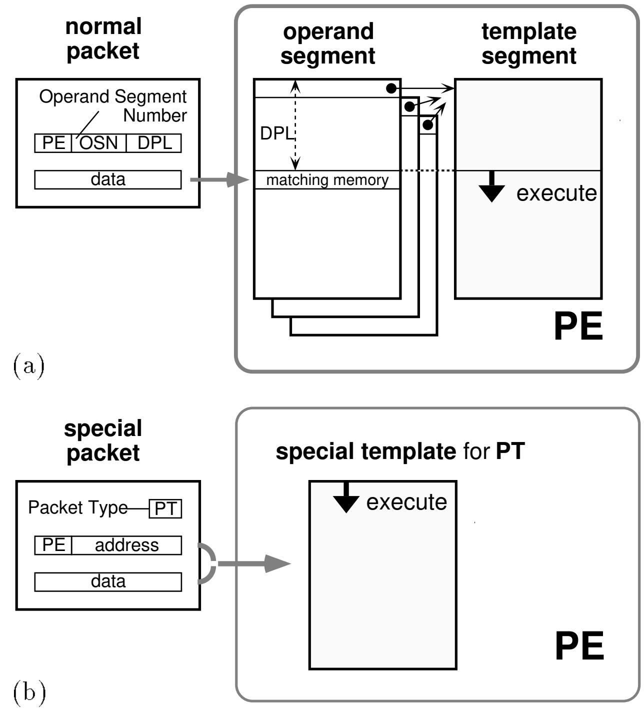


Figure 1: Handling a packet

output a packet into the network directly from the processor.

EM-4 also features a very fast Omega network with support for deadlock avoidance, load balancing, and preservation of transmission ordering. Packet transmission to the adjacent node takes only two clock cycles, and each additional hop takes one clock cycle.

4 The Software Architecture of ABCL/onEM-4

As noted in the last section, EM-4 was originally designed and built to support the dataflow model in which function invocations are side-effect free. We found, however, that EM-4 had far greater potential; since it also supports RISC style imperative features such as memory writes, it is possible to realize side-effectual/state-transitional computation easily and efficiently. We decided to exploit this capability in order to realize concurrent objects which usually have local states.

Since EM-4 is a packet-driven, reactive architecture, it might seem that the reception of a message and the activation of the corresponding method (script) could be carried out by using the direct-matching scheme, and the imperative execution of the method could be done using the instruction sequencing within the SCB. However, these features of the EM-4 hardware are too low level to be employed directly, and an appropriate software architecture must be provided in order to fully realize the concurrent OO computational model in an efficient way. For example, EM-4 requires the exit from SCB when a packet is to be received — as a result, software architecture must guarantee that the object code be restarted subsequently at a proper place after the packet reception.

The basic functionalities that the software architecture must provide are (1) (Remote) Object Creation, (2) Asynchronous Message Passing (messages larger than one packet) and Method Invocation over the network, (3) Intra-node Object Scheduling, and (4) System-wide Garbage Collection. In this section, we describe our proposed software architecture of ABCL/onEM-4, which provides their efficient implementation, fully exploiting the hardware speed; we discuss the rationales and the advantages of the architecture in Section 6.

4.1 Representation of a Concurrent Object in ABCL/onEM-4

In order to implement the concurrent OO-model efficiently, we assume that (1) an object does not span multiple nodes, since the unit of concurrency is each individual object[2], and that (2) there would be multiple objects per each physical node.

In ABCL/onEM-4, each concurrent object is composed of several components. They are: the *object segment*, *reactive text*, and *class-specific text*. The illustrative overview of an object is depicted in Figure 2. We briefly describe the role of each:

- **The Object Segment:** The object segment basically holds the state information of an object, such as the *state variables*, the *temporary variables*, and the pointers for managing its *message queue*. The operand segment described in Section 3 is used for its representation. (When the size of an object becomes larger than the operand segment size (a fixed size, 128 words), the excessive state part of the object is allocated, and a forwarding pointer is placed in the object segment.) The head portion of the object segment are *entries* for realizing the message passing protocols. They are common to all the objects of all classes — the details will be described in the subsequent sections.

  The message queue is a queue of *message boxes* or *mboxes*, which are also fixed in size (32 to 64 words) and allocated from a freelist. An object also maintains a list of *temporary boxes*, which are used to hold the context when its internal routine (i.e., function) is invoked (they are not used for message reception itself). The temporary boxes are allocated from the shared freelist form which message boxes are allocated as well.

  The object segment has a unique address within a single node. Coupled with the unique node ID, the pair constitutes a system-wide unique object ID which serves as a reference to the object.

- **The Reactive Text:** The reactive text is the portion of the object code that appropriately handles each incoming packet to the object entries, and is common to all the objects of all classes. It resides in the template segment, and is pointed to by all the objects in the node.

- **The Class-Specific Text:** The class-specific text is the portion of the object code that realizes the single thread of control of the object. the sequential execution is internally imperative. As the name indicates, there is one class-specific text per each class. It generally performs the followings: (1) initialization, (2) dequeuing of a message from the message queue, (3) method look-up, and (4) method execution. A method is usually executed as a sequence of SCBs, rather than as a single SCB so that packets may be received at the beginning of the SCBs as well as for doing fine-grained scheduling so that hardware packet buffer does not overflow.

[2] Some formal theory of concurrent objects, notably [12], allow for internal concurrency; in this case, an object may span multiple nodes. Whether or not this is useful in practice is the subject of future research.

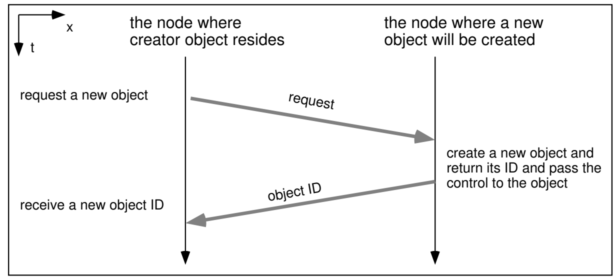


Figure 3: Packet Exchange for Object Creation

4.2 Outline of Object Creation and Message Passing in ABCL/onEM-4

In this section, we outline the software architecture of the packet exchange protocols for implementing fast object creation and message sends. The details of the reactive invocation of the necessary codes are described in Section 4.3.

- **Object Creation:** An object is created in a remote node by sending an object creation request (special) packet to the *object creator* in the remote node. The packet contains the class ID and the reply address. The object creator (1) allocates a single object segment, (2) sends the ID of the allocated object to the reply address, and then (3) passes the thread of control to the newly created object, thereby executing the necessary initialization code within the class-specific text. Figure 3 illustrates the packet exchanging protocol of object creation.

- **Request Message Send:**

  Sending an asynchronous request message to a concurrent object involves much more than just sending multiple packets, because the receiver node must construct a complete message data structure from multiple packets, which are separately sent and are merged with other packets to the receiver node.

  The packet exchange protocol for the message passing in ABCL/onEM-4 proceeds as follows (Figure 4):

  Step 1: A request (special) packet is sent to the receiver node to allocate a message box of a fixed size for storing the message elements.

  Step 2: The message box is allocated in the receiver node, and a (normal) packet containing the message box address is returned to the sender node.

  Step 3: A sequence of (special) packets containing elements of message data are transmitted from the sender to the receiver node. The data are written directly into the message box, as is described in the following sections.

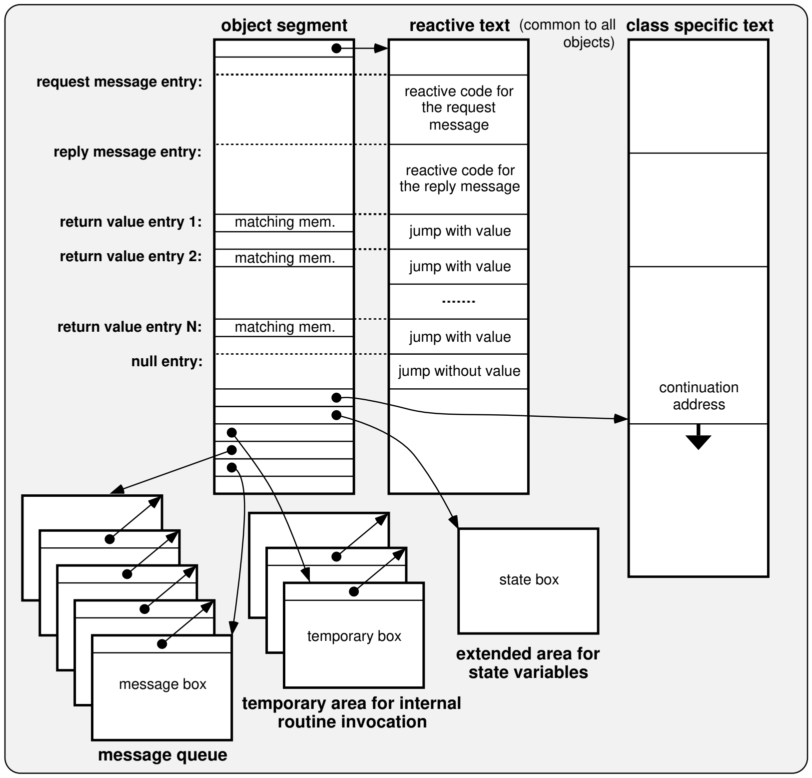


Figure 2: Representation of an object

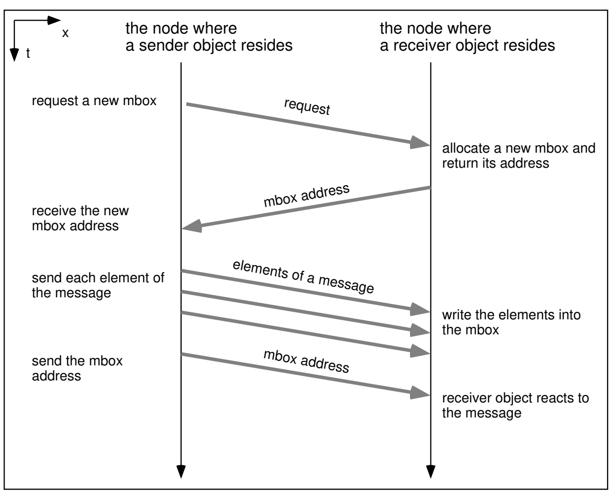


Figure 4: Packet Exchange for a Request Message Send

Step 4: Finally, a (normal) packet containing the address of the message box is sent to the *receiver object* in the receiver node. The packet invokes a proper code so that the receiver object can receive the message and a proper method be invoked if appropriate.

- **Reply Message Send:**

On a now type request message send, the sender specifies a reply-destination of the request message as a pair consisting of the sender object ID and a new reply message box. The reply to a request message is treated analogously to message sends. Since the reply address is already known, and the reply message box will have been preallocated, steps 1 and 2 above are omitted in the usual case.

4.3 Object Entries and Packet Exchange Protocols for Message Sends

ABCL/onEM-4 employs both the normal packet and the special packet on EM-4 for realizing various protocols described above.

Recall that a normal packet invokes the code (with appropriate matchings) with the same displacement in a template segment as pointed to by an operand segment. Since every object segment (which is implemented with an operand segment) points to the reactive text common to all objects, the head portion of an object segment is used as reactive *entries* for receiving packets and invoking various codes within the reactive text (Figure 2). They are (1) the *Request Message Entry*, (2) the *Reply Message Entry*, (3) the *Return Value Entries*, and (4) the *Null Entry*:

- *Request Message Entry* — This is the entry where a packet containing the address of a message box is sent on Step 4 of the message sending described above. The corresponding code in the reactive text executes the following: If the sequential execution of the object is blocked and it is waiting for a new request message, it resumes the execution from the *continuation address* stored in a certain location in the object segment (Figure 2). If not, the message is simply appended to the the message queue.

- *Reply Message Entry* — This is the entry where a packet for a reply message for a now type message is sent. The packet should contain the address of the reply message box in the original sender node. If the sequential execution of the object is blocked and it is waiting for the reply message, it resumes the execution from the continuation address stored in the head of the reply message box. If not, a flag indicating the existence of the reply is set

- *Return Value Entries* — This is a set of two input entries, each of which waits for a pair of packets, one of which contains the returned value from a remote node for some previous request packet (e.g, address of newly allocated objects and/or message boxes), and the other contains the continuation address. The corresponding reactive text contains a 'jump with value' instruction, which jumps to the continuation address specified by the latter packet with the value of former in a register. Note that there are multiple Return Value Entries for requesting multiple resources, e.g., multicasting requires allocation of multiple message boxes on multiple nodes.
- *Null Entry* — This entry is similar to the return value entry, except that only the continuation address is waited for. It is employed for the packet-level, *hardware scheduling* only.

Special packets, on the other hand, are employed to invoke special-purpose *managers* at each node. Recall that a special packet invokes code at some fixed location, and allows the packet to carry two user-defined values instead of one for the normal packet. For ABCL/onEM-4, the manager codes are invoked with special packets. They are: the *object creator*, the *message box allocator*, and the *message writer* (The dark-grey boxes in Figure 5). The former two has the obvious meanings; the message writer is actually a single instruction, which writes an element of a message into a message box.

Given the above entries, here is the more detailed description of the packet exchange protocol for a message send (Figure 5):

Step 1: A request packet is sent to the message box allocator. The packet contains the address of the receiver object (only the destination node information is used here), and the address of the Return Value Entry of the sender object (Figure 5(1)). Before exiting from the SCB the sender self-sends a packet containing the continuation address within the method where the evaluation of the arguments of the message is performed.

Step 2: A message box is allocated in the receiver node from the freelist by the mbox allocator, and its address is sent to the Return Value Entry of the sender object (Figure 5(2)). This packet is matched with the self-sent packet of Step 1, invoking the argument evaluation/sending from the continuation address.

Step 3: As each argument is evaluated, a sequence of special packets containing both the elements of message data and the address within the message box where the data is to be stored, are transmitted from the sender object to the message writer of the receiver node (Figure 5(3)). This process may be interrupted by nested message sends, as is described in Section 6. The message writer writes the data directly into the specified address in the message box.

Step 4: Finally, a packet containing the address of the message box is sent to the Request Message Entry of the receiver object (Figure 5(4)). As described above, this triggers the execution of appropriate code so that the receiver object can either invoke a specified method or just enqueue the message depending on the object state.

4.4 Scheduling of Objects

The scheduling of objects within a node is performed using two types of schedulers. One is the *hardware scheduler*, and the other is the *software scheduler*. Both employ the packet-driven architecture of EM-4.

The hardware scheduler directly utilizes the hardware packet buffer as the scheduling queue. When scheduling of an execution of code is needed, a packet containing a continuation address is sent to one of the object entries, typically a Return Value Entry or the Null Entry. This packet is buffered in the hardware packet buffer, which will eventually be scheduled for execution.

The software scheduler is similar to more conventional schedulers but nevertheless is packet-driven. That is to say, since it too is a code that must be invoked via the packet-driven mechanism, its address must be enqueued in the hardware packet buffer. When it is scheduled, it re-enqueues itself in the hardware packet buffer for subsequent re-scheduling of itself and then dequeues one of the objects it holds in its software scheduling queue and transfers control to the object. When an object is to be a subject of software scheduling, the software scheduler enqueues the object in the software scheduling queue and then makes an exit from the SCB, thereby causing a context switching in the hardware scheduler.

The combination of hardware and software schedulers has the following advantages. High priority tasks, such as reception of a reply message, are scheduled by the hardware scheduler for fast dispatching. Lower priority tasks are managed and scheduled by the software scheduler, which (1) prevents the hardware buffer overflow[3], and (2) allows for flexible scheduling policies other than the round-robin.

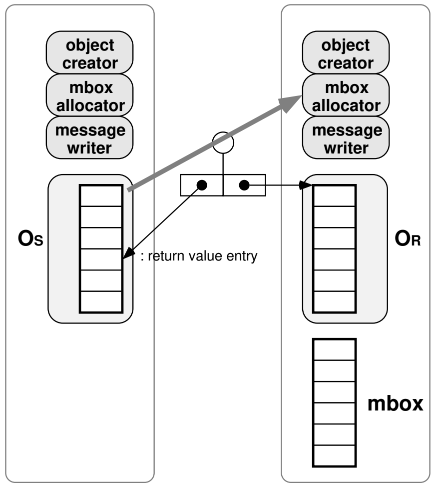
4.5 Garbage Collection

Garbage collection is performed based on the marking algorithm using virtual global snapshot and object coloring. The algorithm is similar to the one employed by the POOL-T language[3]. The details of our GC scheme will be reported by a separate paper.

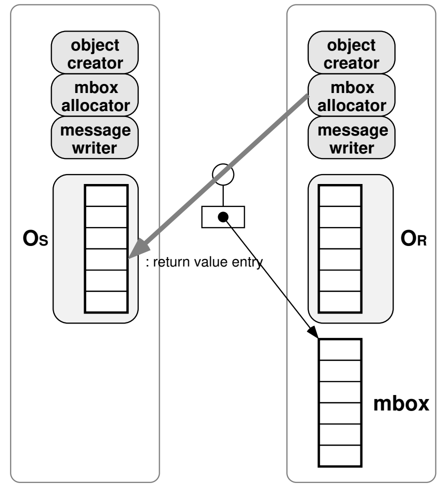
5 Performance Measurements of ABCL/onEM-4

Currently, our ABCL/onEM-4 cross-compiler is partially complete. The measurements in this section were taken with partially hand-compiled code running on the actual EM-4 hardware[11] (80 PEs, 12.5MHz clock). The measurements were also confirmed to coincide with the simulator results.

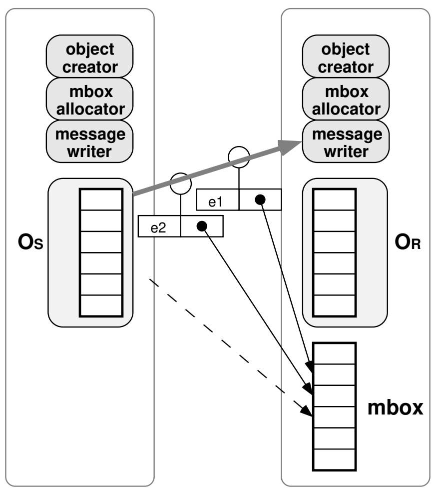
[3]Although hardware buffer overflow is automatically handled by the EM-4 hardware, it incurs significant overhead because the overflowed elements must be written to/read from the main memory.



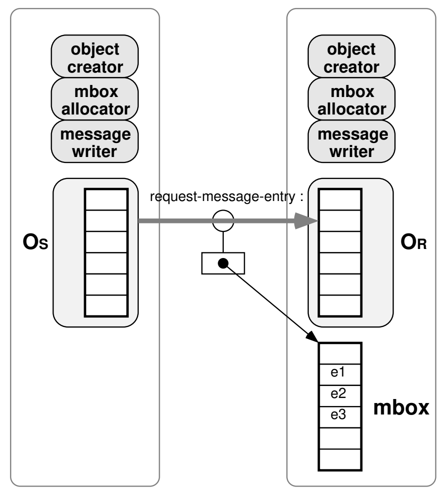
(1) Step 1: requesting a new mbox



(2) Step 2: returning a new mbox



(3) Step 3: writing elements into mbox



(4) Step 4: sending the mbox address

Figure 5: The details of packet exchanges for a message send

5.1 Latency Evaluation – Counters

In order to measure the latency of object creation and message send/reply cycle, our first benchmark program (Figure 6) creates a double loop of closed Hamilton path among the 80 nodes (160 objects total), and sends a message through the loop. The `:create-count` message creates the path as the messages are sent, and the `:count` message utilizes the existing objects created by `:create-count`. Since the messages are sent using the now type (RPC-style) message passing, the message sendings are serialized, i.e., there is only one message being sent throughout the entire system at any given moment.

Table 1 shows the results of the benchmark. When the path is created dynamically using the `:create-count` message, the total execution time is 1.665 milliseconds (20810 clock cycles), or approximately 10.41 μseconds (130 clock cycles) per object creation/message send/reply. When the path already exists, the execution time is 1.434 milliseconds (17922 clock cycles), or 8.96 μseconds (112 clock cycles) per message send/reply. Since the difference between the two is the creation overhead of 160 objects along the path, the time required for object creation is 1.44 μseconds (18 clock cycles) per object.

5.2 Throughput Evaluation – Fibonacci Numbers

The previous program only measures the latency of message processing; as a result, there was little concurrency involved. We next measure the number of method invocations possible throughout the entire system per each time unit. The example is a tree computation induced by the calculation of Fibonacci numbers (Figure 7). The objects of class `fib` comprises the tree, either becoming an interior node or a leaf node depending on the argument of the message. The leaf node immediately returns a reply message (the value 1). The interior node (1) creates two sibling objects, (2) multicasts the `:calc` message to the siblings using the now type message send, and waits for their reply, then (3) computes the new value returns the result to its parent node. Thus each object is first (a) created, (b) receives a request to calculate and (c) answers a reply Here, for convenience, we count the above process (a) – (c) as one *task*. (We also make a note that this program is solely

```
[class counter ; Class definition
  (state next-obj) ; An state variable to hold the name of object on next node
  (script
   (=>[:create-count (integer n)] ; When a message :create-count arrives,
      (if (<= n 1)
          ![1] ; returns 1 ('!' is the reply construct)
          (progn [next-obj := [create counter]] ; A newly created counter object is set to next-obj
                 ![[next-obj                    ; Now type message sending
                            <== [:create-count (- n 1)] [(integer)]]])))
   (=>[:count (integer n)]          ; When a message :count arrives,
      (if (<= n 1)
          ![1]
          ![[next-obj ; employ existing path
                   <== [:count (- n 1)] [(integer)]]]))))]

[object main
 (state counter-object)
 (script
  (=>[:run (integer n)]
     [counter-object := [create counter]] ; creation of initial object of path
     (print [counter-object <== [:create-count n] [(integer)]])
     (print [counter-object <== [:count n] [(integer)]]))))]

;;; creates double loop of Hamilton path
[main <= [:run 160]] ; past type message sending
```

Figure 6: Message loops through a Hamilton path

	creation-request-reply cycles	request-reply cycles
Total execution time (160 objects)	1.665 msec (20810 clocks)	1.434 msec (17922 clocks)
Average execution time per one cycle	10.41 μsec (130 clocks)	8.96 μ sec (112 clocks)

Table 1: Execution time for remote-creation/send/reply cycles

for performance measurements only, and by all means NOT the fastest way to compute the Fibonacci number with an OOCP language.)

We computed the Fibonacci number for $\mathtt{n} = 3 \ldots 20$ using this program. Figure 8(1) shows the execution time of the program versus the total number of tasks processed. The graph indicates that over 13000 tasks are performed in 4.1 milliseconds, or approximately 3 tasks per 1 μsecond. In order to see whether this is the maximum, we consider the second graph (Figure 8(2)) which shows the task processing rate (tasks/millisecond) against the total number of tasks. As we can see, the processing rate saturates between 1000 to 10000 tasks, or $\mathtt{n} \geq 15$ at the above rate.

6 Discussions and Comparison to Previous Approaches

Our ABCL/onEM-4 software/hardware architecture offers substantial performance improvements over the previous implementation architectures of OOCP languages, such as the J-Machine[5] and A-NET[4]. Here, we would like to clarify the differences between our approach and the previous approaches, specifically how we reduce the overhead of message sends with appropriate software/hardware architectural combination.

In the previous approaches, the execution time was dominated by the overhead of message processing. To reduce it, we must improve the speed of context switching and message reception/dynamic method invocation.

6.1 Overhead Reduction in Packet Dispatch/Context Switching

Much of the low-level provisions for fast packet dispatching and context switching is provided by the EM-4 hardware. In addition, the compiler supports fast context switching by minimizing the context.

6.1.1 Fast Dispatch and (Pipelined) Zero-Overhead Context Switching

The EM-4 hardware features a very low latency from the time the packet is received until the code is actually invoked. As a result, the reactive style of concurrent object-oriented computation can become feasible. A very high performance is realized with the direct matching scheme and the reduction of pipeline stages. Typical latency is approximately 3 clock cycles, or 240 ns. Special packets are processed even faster, with typical pipeline latency of 2 clocks, or 160ns.

EM-4 also features zero-overhead context switching. The context switching occurs when the CPU exits from the SCB and receives a packet. There is no context switching overhead involved owing to *pipeline fusioning* of the *normal dataflow pipeline* and the *strongly connected pipeline*[13].

6.1.2 Flexibility in Amount of the Context to be Preserved

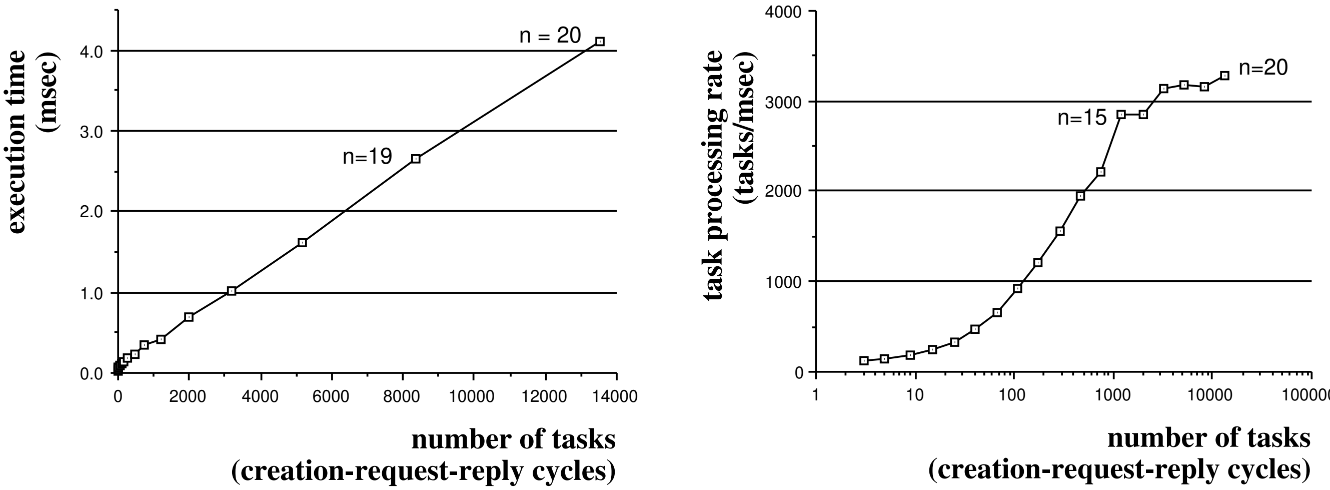
Some machines have (1) provided support for automatic preservation of register contexts, and/or (2) decreased the number of general registers in order to reduce the overhead of context switching. For example, in J-Machine, there are only 4 general registers, which are automatically preserved on certain traps.

By contrast, EM-4 has 10 general registers (16 registers total), allowing most of the computation to be performed in a RISC-like manner. EM-4 does not provide automatic register context preservation; thus, we must resort to software for register preservation. In ABCL/onEM-4, however, we can reduce the amount of necessary context information to be preserved to the minimum, since we know when the context switch will occur due to the mutual exclusiveness of the strongly

```
[class fib
  (script
   (=>[:calc (integer n)]
      (if (<= n 2)
          ![1]
          ![(+ [[create {fib fib}] ; multi creations
                <== {[:calc (- n 1)] [:calc (- n 2)]} ; multicast now type message sends
                     {[(integer)][(integer)]}])]))))]

[object main
 (script
  (=>[:run (integer n)]
     (print [[create fib] <== [:calc n]]))))]
```

Figure 7: Simple-minded Fibonacci numbers with object creation



(1) Execution time

(2) Throughput

Figure 8: The result of the throughput measuring program for ABCL/onEM-4

connected mode. The minimum context information needed to be preserved is the address of the object segment and the continuation address within the class-specific text[4]. Whether other registers are saved or not upon a exit from the SCB is decided by the flow analysis performed by the compiler.

6.2 Overhead Reduction in Remote Message Passing

Another major difference between ABCL/onEM-4 and previously proposed architecture-supported OO-language implementations such as Concurrent Smalltalk on the J-Machine[8] and A-NET[4] is that we attempt to eliminate the memory read/write/copy overhead of the outgoing and incoming messages as much as possible. That is to say, whether messages will require buffering or not could have significant impact on performance due to overheads such as copying and management of the buffers.

Here is the summary of the message sending activity of ABCL/onEM-4:

- The sender node reserves the message box on the receiver node.
- The sender evaluates the arguments of the message. As each argument is evaluated, its packet is sent to the proper (address) location in the message box in the receiver node. The evaluation and message send are pipelined.
- The message box address is sent to the receiver object.

By contrast, here is the summary of the message sending activity of the J-machine:

- The `SEND` instruction loads message words into the FIFO message buffer. When the message is complete (indicated by the `SENDE` instruction) or the buffer is full, the message is sent to a specified node.
- When the message reaches the receiver node, it is automatically enqueued in one of the two circular buffers (one for each priority level) in that node. these circular buffers are also used as scheduling queues.

Given the above characterizations, we now discuss the advantages of our scheme.

6.2.1 Efficient Management of Message Buffers with Freelists

In our scheme, the message buffers are managed with the freelist. The message boxes are efficiently allocated from/deallocated to the freelist. Recall that the size of message boxes is fixed. This of course incurs storage overhead — a space/speed tradeoff. In practice, all is not wasted, since the unused portion of a message box is used for temporary variables and scratchpads.

Compared to using the hardware circular buffer as was adopted by the J-machine, our scheme using the

[4]In fact, as we have seen in Section 4, these are not only stored in the local store, but may also exist within packets being transmit.

freelist has a great advantage that *copying of queued messages is unnecessary.* By Contrast, For hardware circular buffer, the message data will have to be copied and removed from the circular buffer into some external buffer in the following cases:

- *When scheduling other than first-come-first-serve is preferable.* In many classes of problems, such as discrete event simulation algorithms[10] and search problems, proper scheduling of message processing is essential in order to achieve good performance.
- *When an object cannot process the received message immediately.* This happens when the object is waiting for the reply message of the previously sent request message. Consequently, the messages which are scheduled before the reception of the reply message must be copied from the circular buffer. For practical complex parallel programs, this is often the case.

6.2.2 Reservation of Message Boxes

In our scheme, the sender reserves the message box prior to the actual sending of the message data. Although the packet transmissions to allocate a message box may result in some overhead, the turnaround overhead is not significant owing to the fast network architecture.

Without such reservation, the receiver node has to allocate the message buffer as fast as the network speed. Usually this requires some special hardware to allocate the message buffer, and it is difficult for such hardware to realize the complex allocation scheme such as one using the freelist. In our scheme with the reservation, allocation can be done in software, which performs the complex allocation without such hardware.

The reservation is useful not only for receiving the message but also for sending the message. That is to say, since the sender node can obtain the address of message box prior to the actual sending of the message data, the per-packet, pipelined and interruptible sending of the message data becomes possible as is described next.

6.2.3 Sending of Message Elements with Per-Packet Communication

EM-4 allows per-packet specification of the address to which the packet data is to be written. Once the sender gets the address of the message box, it can emit the packets as the message expressions are evaluated. As a result, (1) the latency of sending of actual message data can be reduced because most of the message elements can be sent into the network before the sender node gets the complete elements of the message, and (2) message elements can be sent in an arbitrary order at arbitrary times; in fact, sending of a single message (consisting of multiple packets) could be interrupted in its midst, and interleaved with other message sends.

On the other hand, in some architectures such as the J-Machine, since the location where the message will be stored is not determined upon its sending, once the receiver node is specified, the rest of the message elements must be sent continuously. This will reduce the size of messages, but has the drawback that the message sending cannot be interrupted.

Our claim is that, non-interruptible messages decreases throughput by requiring local memory accesses. Let us consider the following example:

```
[ObjA <= [:MsgA (- n 1) (Fun x y)
                        [ObjB <== [:MsgB]]]]
```

The above message sending expression contains a function call and an embedded message sending expression as arguments. In ABCL/onEM-4, as soon as each argument is evaluated, it is immediately sent to the node of the target object (*ObjA*). The evaluation of the second argument (function call) and the third argument (message send to *ObjB*) could or will effectively interrupt the sending of the outer message, but this will not incur any overhead as explained above. By contrast, in the J-machine, the arguments must be saved (or, buffered) into memory as they are evaluated, because of possible interruption. At the end of the evaluations of all the arguments, the results must be re-read from memory when they are actually sent. Altogether, the benefits of direct message sending from the hardware processor would be lost here; and extensive flow analysis performed by the compiler will only serve as a minor solution.

6.3 Comparison to Implementation of Procedural Languages

There is another language implementation work in progress on EM-4[14]; there, a subset of the C language augmented with message passing libraries is being developed. We will do a brief comparison and argue why our approach is faster.

Aside from the fact that procedural languages require extensive flow analysis to extract the inherent parallelism, library-based message sending has the drawback of requiring extraneous memory accessing: according to the language definition of C, function call is call-by-value, i.e., all the arguments must be evaluated before invoking the function. This requires that the evaluated arguments to be stored into memory, then re-read in order to perform a message send. This nullifies the benefit of being able to perform fast message passing directly from the hardware processor.

6.4 Summary of our Comparison to Previous Approaches

To summarize, our results in Section 5 indicate that, concurrent message processing for concurrent objects can be significantly improved over the previous approaches, and as a result, the OOCP languages could serve well in a massively parallel environment. We believe that this claim holds even for the small examples in this paper. For example, by the numbers given in [7], the total number of machine cycles for object creation followed by a message send and its reply is estimated to be well over 200 cycles for the J-Machine, whereas it is approximately 130 cycles in our case. And, for the reasons we gave in this section, we further believe that the difference in actual throughput should increase in even larger, real-life applications.

7 Conclusion

Parallel architectures with strong support for programming languages have been studied for a long time. Yet, architecturally novel computer organizations often lacked OO views. The hardware, EM-4, envisioned so as to service homogeneously interconnected dataflow/control-flow computation, has proven to be extremely well-suited for concurrent OO computation when combined with our software architectural technologies. As we have indicated, compared to our other implementation work of ABCL on a more conventional multicomputer based on conventional SPARC chips (Fujitsu AP1000[9]), we have been able to achieve an order of magnitude improvement in inter-node message passing speed (approximately 35 μseconds vs. a few μ seconds). We have also been able to surpass another concurrent-OO software/hardware architecture based on microcodes by two-orders of magnitude.

The principle behind our software/architecture is very similar to that of the current RISC technology — rather than to provide very complex hardware for message passing that could take hundreds of machine cycles with microcodes, we base ourselves on a low-level reactive architecture with packet latency and throughput comparable to intra-node processing, and then build an appropriate software architecture with language and compiler support. Architectures such as J-Machine share our views in many ways, but as discussed in Section 6, our software/hardware architecture is cleaner/faster by comparison, allowing us to avoid much of the memory operations, which results in considerable overhead when the latency of hardware packet architecture is minimized. We hope that the techniques and results we have described in this paper will pave the way towards development of even more efficient implementation technologies for OOCP languages and languages of other computational paradigms.

Acknowledgements

We would like to express our sincere thanks to Dr. Toshitsugu Yuba and Dr. Toshio Shimada at ETL for allowing us to use EM-4. We are also very much grateful to Dr. Yoshinori Yamaguchi, Dr. Shuichi Sakai, Dr. Mitsuhisa Sato and Dr. Yuetsu Kodama at ETL. Their technical helps were the major source of encouragements and numerous technical discussions with them truly inspired us. Finally, we thank the members of the ABCL project group at our laboratory for their daily discussions and assistance.

References

[1] Gul Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[2] Pierre America and Jan Rutten. A layered semantics for a parallel object-oriented languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. of REX/FOOL, Noordwijkerhout, The Netherlands*, volume 489 of *Lecture Notes in Computer Science*, pages 91–123. Springer-Verlag, May/June 1990.

[3] Lex Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijiman, and P. C. Treleaven, editors, *Proc. of PARLE, Eindhoven, The Netherlands*, volume 259 of *Lecture Notes in Computer Science*, pages 75–93. Springer-Verlag, June 1987.

[4] Takanobu Baba, Tsutomu Yoshinaga, Tohru Iijima, Yoshifumi Iwamoto, Masahiro Hamada, and Mitsuru Suzuki. A parallel object-oriented total architecture: A-NET. In *Proc. of Supercomputing'90, New York*, pages 276–285, November 1990.

[5] William J. Dally. The J-machine: System support for Actors. In Carl Hewitt and Gul Agha, editors, *Knowledge Processing: An Actor Perspective.* The MIT Press, 1989.

[6] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proc. of ECOOP'91, Geneva, Switzerland*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[7] Waldemar Horwat. Concurrent smalltalk on the message-driven processor. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[8] Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proc. of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 101–109, June 1989.

[9] Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, and Sadayuki Kato. An architecture of highly parallel computer AP1000. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16, May 1991.

[10] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[11] Yuetsu Kodama, Shichi Sakai, and Yoshinori Yamaguchi. A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation. In *Proc. of InfoJapan '90*, pages 291–298, 1990.

[12] José Meseguer. A logical theory of concurrent objects. In *Proc. of OOPSLA/ECOOP'90, Ottawa, Canada*, pages 101–115, October 1990.

[13] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[14] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[15] Etsuya Shibayama. *An Object-Based Approach to Modeling Concurrent Systems.* PhD thesis, Department of Information Science, The University of Tokyo, 1991.

[16] Etsuya Shibayama and Akinori Yonezawa. Distributed discrete event simulation in ABCL/1. In Akinori Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System*, chapter 9. The MIT Press, 1990.

[17] Masahiro Yasugi and Akinori Yonezawa. An object-oriented concurrent algorithm for N-body problem. In *Proc. of JSSST 8th Annual Convention, Sapporo, Japan*, pages 405–408, September 1991. (in Japanese).

[18] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[19] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. of ACM Conference on OOPSLA*, pages 258–268, 1986.

[20] Akinori Yonezawa and Ichiro Ohsawa. Object-oriented parallel parsing for context free grammars. In Akinori Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System*, chapter 11. The MIT Press, 1990.