Adaptive Objects for Concurrent Accesses

Masahiro Yasugi
Department of Communications and Computer Engineering
Graduate School of Informatics, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
e-mail: yasugi@kuis.kyoto-u.ac.jp

Shigeyuki Eguchi
Graduate School of Science and Technology, Kobe University

Kazuo Taki
Department of Computer and Systems Engineering, Kobe University
1-1 Rokkodai Nada Kobe, JAPAN 657-8501

ABSTRACT. *The efficiency of parallel execution is often limited by the presence of bottlenecks in a small region of the entire system. A major source of such bottlenecks is the mutual exclusion (serialization) of concurrent accesses to a single object. Some bottlenecks can be eliminated (1) by relaxing the rule of mutual exclusion (e.g., by providing relaxed mutual exclusion which permits simultaneous read-only accesses to an object) or (2) by introducing immutable objects and permitting local access to their copies. In the conventional object-oriented parallel programming languages, these techniques are applied only at the compile time, which makes lots of bottlenecks in the practical parallel applications still remain. In this paper, we propose dynamic (runtime) method replacement which provides us adaptive objects for concurrent accesses, where we can increase the number of read-only methods and the immutable part of the objects dynamically. The proposed scheme is realized by encapsulated message renaming, where the mechanism of class inheritance is naturally extended. This paper also presents the results of the performance measurements on a shared memory parallel architecture and a distributed memory parallel architecture, which indicate the effectiveness of our approach to the bottleneck elimination.*

1 Introduction

The benefit of parallel processing is the speed-up on the real parallel computers. However, the efficiency of parallel execution is often limited by the presence of bottlenecks in a small region of the entire system. In the object-oriented parallel computing, a major source of such bottlenecks is the mutual exclusion (serialization) of concurrent accesses to a single object (such as a root object of a tree structure).

Some bottlenecks can be eliminated (1) by relaxing the rule of mutual exclusion (e.g., by providing relaxed mutual exclusion which permits simultaneous read-only accesses to an object) or (2) by introducing *immutable* objects and permitting local access to their copies. The relaxation of mutual exclusion can be seen in [TY 95]. The introduction of immutable objects can be seen in [SAR 93, TMY 94].

In the conventional object-oriented parallel programming languages, however, these techniques are applied only at the compile time, which makes lots of bottlenecks in the practical parallel applications still remain. For instance, when the application employs a network of objects (e.g., a binary search tree and a neural network) which is constructed by updating the states of objects during the execution, the compile time approach does not work since the bottleneck can be eliminated only after the construction.

In this paper, we propose *dynamic (runtime) method replacement* which provides us adaptive objects for concurrent accesses, where we can increase the number of read-only methods and the immutable part of the objects dynamically. This feature is implemented and examined in our object-oriented parallel programming language OPA[YT 96].

OPA (Object-oriented language for PArallel processing) is a parallel extension of Java[AG 96]; we remove specification about threads and monitors from Java then add new constructs for structured parallel processing, synchronization and relaxed mutual exclusion to it. Its design is intended to realize both ease-of-use and high performance of parallel processing. The proposed dynamic method replacement is realized by encapsulated message renaming, where the mechanism of class inheritance and polymorphism is naturally extended.

We are implementing OPA on a shared memory parallel computer. On shared memory architectures, special attention must be paid to the development of the implementation scheme for dynamic method replacement, since the method replacement and the method lookup themselves require mutual exclusion. In particular, the lookup of a replaceable method should be performed without costly lock/unlock operations if the extracted method is a read-only method.

We are also implementing OPA on a distributed memory parallel computer. On distributed memory architectures, it is easy to replace methods without lock/unlock operations. We use a *proxy* of an actual object if the actual object exists on a remote node. A message passing to the proxy causes either of (1) a remote method invocation to the actual object or (2) a local method invocation on the proxy. The latter is valid for the method which only reads immutable part of an object. Our dynamic method replacement contributes to increasing the latter case.

The rest of the paper is organized as follows. In Section 2, the brief overview of our object-oriented parallel language OPA is described. We illustrate the design of dynamic method replacement in Section 3.

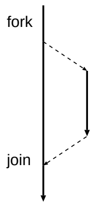

Figure 1: Fork-join parallel

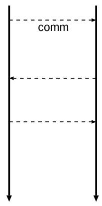

Figure 2: Cooperative parallel

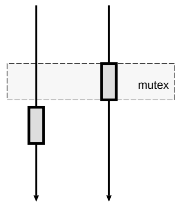

Figure 3: Mutually exclusive parallel

A free method detection algorithm is presented in Section 4. The implementation on a shared memory parallel computer is presented in Section 5, and that on a distributed memory parallel computer is presented in Section 6. The results of the performance measurements on both parallel architectures are presented in Section 7. In Section 8, we discuss the related work and the effectiveness of our approach.

2 Object-oriented parallel language OPA

This section presents the brief overview of our object-oriented parallel language OPA. OPA is a parallel extension of Java language; we remove specification about threads and monitors from Java then add new features for (1) structured parallel processing, (2) synchronization and (3) relaxed mutual exclusion. In the subsequent subsections, we will describe each of these. Note that we use Java as base language because of its simple and clear semantics; our proposed scheme can also be applicable to the other object-based sequential languages.

The design of OPA is intended to realize both ease-of-use and high performance of parallel processing. OPA divides the patterns of parallel processing into three types according to the relation among threads, namely fork-join parallel (Figure 1), cooperative parallel (Figure 2), exclusive parallel (Figure 3). OPA supports irregular parallelism with dynamically forked threads; however, the fork and join of threads are performed in a structured manner. Objects are mainly utilized to represent complex data structure, where object can hold a reference to another object. Each object does not have a thread of control; as in sequential object-oriented languages, the method lookup and the subsequent method invocation on an object are performed by the thread that sends a message to the object. The mutual exclusion is necessary for concurrent accesses (message passings) to an object to read/write the object's state consistently. The objects are also used for synchronization among threads performing message passing to it; such synchronization (e.g., cooperative parallel) is recommended only when it can not be captured by the structured fork-join parallelism.

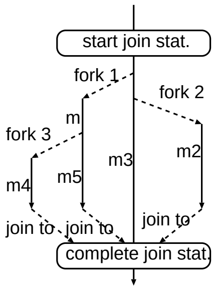

Figure 4: fork-join parallel by join statement and par statement

2.1 Structured fork-join parallel

The structured fork-join parallel is another powerful feature of OPA. While this feature is not directly related to the topic of this paper, it should be noted that we have several reasons to make threads independent of objects. First, as described later, the separation provides some quick and consistent method invocation on an object especially by permitting simultaneous read-only accesses to an object, although we can regard such method as being performed by the object's thread which is created in response to message acceptance. Second, the fork-join parallel can be well structured, when we assign a thread not for an object but for a *task*.

The task may be executed by a single thread, where the thread may execute methods of a number of objects via message passing. The task can be divided into two or more subtasks; such decomposition will repeat recursively until we get primitive operations. If a divided subtask can be executed in parallel, we can fork and join a new thread to assign the parallel subtask to the thread.

We introduce join statements, syntactically denoted by "`join` *statement*," to join a number of threads; by "`join` *statement*," the *statement* are executed with the current thread and the new threads created with the execution of the *statement* are joined with the completion of the join statement. Par statements, syntactically denoted by "`par` *statement*," are used to fork new threads; by "`par` *statement*," a thread is forked to execute the *statement*.[1] For instance, the statement:

```
join { par o.m(); par o2.m2(); o3.m3(); };
```

forks two threads to execute `o.m()` and `o2.m2()` respectively and executes `o3.m3()` with the current thread then joins the termination of the forked

[1]In the current version of OPA, the *statement* should be a method call and the actual parameters are evaluated with the current thread. For general statements, the original thread and the forked thread share the environment; in OPA, the variables that the forked thread may access to are copied into the local state and the update of variables by the forked thread is reflected at the join point.

threads (see Figure 4). Furthermore, if a forked thread forks another new thread, the new thread will be also joined. For example, if the definition of m, which is executed by the first forked thread, is as follows:

```
m(){ par o4.m4(); o5.m5();}
```

the thread to execute `o4.m4()` is also joined at the end of the join statement.

When a thread is forked, its join-destination is automatically determined as the inner most join statement within which the thread is forked. This rule eliminates undesirable sources of bugs, namely explicit "join"s or "join to"s, just like explicit "label"s or "goto"s for the structured loop/conditional statements.

2.2 Synchronization

In this paper, by synchronization, we mean that a thread suspends its execution until a resumption condition is satisfied. In OPA, most patterns of synchronization can be captured by the structured fork-join statements. However, the structured approach does not cover cooperative parallel processing in which the related threads synchronize/communicate with each other in the course of their execution (i.e., without termination of their execution). For instance, in producer/consumer relationship, the producer and the consumer communicate in a pipelined manner.

In OPA, objects are employed for synchronization among cooperatively parallel threads which perform message passing to the shared objects. OPA provides useful classes for synchronization, such as I-structure (I-store) class, FIFO queue class, and barrier synchronizer class. For example, an I-store object suspends all `get` accesses to it until a `put` access instantiates its value.

To program classes for synchronization, Java employs Monitor[BFC 95] with `wait()` operation (to enqueue the current thread in the object's waiting queue and unlock the object), `notify()` operation (to dequeue a thread from the object's waiting queue and make it runnable) and `notifyall()` operation (to perform `notify()` operation for all threads in the object's waiting queue).

However, the Java's approach has two problems. First, while a thread is suspended by a `wait()` operation on an object, another thread may invoke a method on the same object. Since the `wait()` operation appears in the middle of the invoked method, it is more difficult to keep the state of the object consistent. Let **B** be the beginning of a method, **E** be the end of a method, **S** be the suspension at a `wait()` operation and **R** be the resumption from a `wait()` operation. The programmer should keep the state at **E** (or **S**) consistent with the state at **B** (or **R**); thus, the programmer should keep track of 4 possible combinations, namely **E**-**B**, **E**-**R** **S**-**B** and **S**-**R**, whereas only **E**-**B** needs attention if there is no `wait` operation. Secondly, Java does not employ condition variables; only one waiting queue is associated with an object. Therefore, to waken a specific thread prior to other threads waiting their own conditions, we have to repeat the `wait()` operation until its waiting condition is satisfied and use `notifyall()` at every point where any of the waiting conditions may become true. When the number of threads in the waiting queue increases, this scheme incurs significant overhead.

In OPA, to program classes for synchronization, we first combine the prepared useful classes. For further programming, we can employ extended dynamic method replacement: if a method has been replaced with a special keyword, `suspends`, threads which try to invoke the method must suspend their execution until the method is replaced with a normal method. Since the suspension occurs only before a method is invoked and a waiting queue is associated with each of replaceable methods on an object, the above problems are not the cases with our approach. The detail of dynamic method replacement is described in Section 3.

2.3 Mutual exclusion

In the object-oriented parallel computing, mutual exclusion (serialization) is necessary for concurrent accesses (message passings) to an object to read/write the object's state consistently. As was described in Section 1, such mutual exclusion can be relaxed by permitting simultaneous read-only accesses to the object or can be eliminated by introducing immutable objects.

The use of immutable objects can be easily extended to that of partly immutable objects: if a method only reads immutable part of an object, the method may be invoked without mutual exclusion. Let us call such method *free*. On distributed memory architectures, exploiting free methods also allows the language system to make a closure for a free method and distribute copies of the closure to the other nodes to permit local invocation on the copy. The effect of using free methods is significant, since using local copies almost perfectly eliminates bottlenecks.

When the dynamic method replacement is employed, however, it is difficult for the compiler to confirm that a method is free, since every possible combination of methods must be examined. Thus, we must assume that the compiler may fail to detect free methods within a limited time and space complexity. We should not change the semantics of language due to the limitation of the compiler; we can not employ free methods directly in the language.

Instead of free methods, non-blocking read-only methods, which are proposed in [TY 95] for deadlock avoidance, can be employed in the language. A non-blocking read-only method may read mutable part of the object but it can be executed without infinite blocking. This is realized by updating the object state atomically. Now, we can exploit free methods as optimized implementation of non-blocking read-only methods without change of the language semantics.

To incorporate non-blocking read-only methods, OPA uses relaxed mutual exclusion for the methods defined with keyword `instant`. The compiler automatically divides instant methods into read-write (RW) type and read-only (RO) type according to their access to the object. The only literal accesses

```
class BinTree{
  internal int key, value;
  internal BinTree left = null, right = null;
  public BinTree(int k, int val){
    key = k;
    value = val;
  }
  public instant int search(int k, int v){
    if(k == key)return value;
    else if(k < key){
      if(left != null)return left.search(k, v);
      else return v;
    }else{
      if(right != null)return right.search(k, v);
      else return v;
    }
  }
  public replaceable instant void insert(int k, int val){
    if(k < key){
      if(left != null)left.insert(k, val);
      else{
        left = new BinTree(k, val);
        if(right != null)setmethod(insert, insertF);
      }
    }else{
      if(right != null)right.insert(k, val);
      else{
        right = new BinTree(k, val);
        if(left != null)setmethod(insert, insertF);
      }
    }
  }
  internal instant void insertF(int k, int val){
    if(k < key)left.insert(k, val);
    else right.insert(k, val);
  }
}
```

Figure 5: Example of dynamic method replacement (binary tree dictionary)

are checked (i.e., the access performed by the another method invoked within the current method is ignored). For instance, the `insertF` method and the `search` method in Figure 5 are of RO type and the `insert` method is of RW type. The `internal` keyword in Figure 5 is appended to OPA language to indicate that the field or method is only accessed via explicit or implicit `this` reference.

Both RO and RW methods read the necessary variables of the object into implicit local variables atomically at the beginning of the method, while an RW method writes the values of local variables into the object atomically at the point where the rest of the method execution no longer updates the local variables. The compiler automatically determines the update point with flow analysis.

The RO method can read the state of an object, even if an RW method is running on the object. The RO method is only prohibited from reading the state which is partly updated. For an RO method and an RW method, respectively, the message passing is performed with the following steps:

RO method: (RO1) lookup the method. (RO2) invoke the method. (RO3) atomically read the object's state into local variables. (RO4) execute the rest of the method. (RO5) return.

RW method: (RW1) lookup the method. (RW2) invoke the method. (RW3) atomically read the object's state into local variables. (RW4) execute the method until the atomic update. (RW5) atomically write the updated local data into the object. (RW6) execute the rest of the method after the atomic update. (RW7) return.

Let *RW section* be the section from RW1 to RW5, *RO section* be the section from RO1 to RO3, *W section* be the section of RW5. The mutual exclusion is necessary between an execution of RW section and another execution of RW section (Let us call this mutual exclusion *RW-RW mutex*). The mutual exclusion is also necessary between an execution of RO section and another execution of W section (Let us call this mutual exclusion *RO-W mutex*). RO1 and RW1 is included in RO section and RW section respectively, since that is required for the dynamic method replacement. The mutual exclusion between the execution of W section and another execution of W section is automatically realized as a part of RW-RW mutex. By "mutual exclusion," we mean that the two or more threads may not be executed in parallel; that is, executions of high-level operations for different threads may not be interleaved. This means that the implementation-level operations may be executed in parallel unless it changes the semantics of higher-level mutual exclusion.

With the instant methods, not only simultaneous read-only accesses to the object are permitted, but also those read-only methods can be executed in a non-blocking manner.

3 Dynamic method replacement

To confirm that a method is free, it is necessary to confirm not only that the method does read-only access to the object but also that the read part will be never written by the other methods. The latter condition is usually satisfied at least only after the initialization: most methods are not free when the constructors are initializing the object. The constructors will not be invoked once the object creation is finished; some methods become free after the creation.

In practical applications, the object often has some RW methods which do write access to the object; however, after they are invoked finite times (usually for setting up the state of the object), it is often the case that (a) they are no longer invoked or (b) they no longer do write access to the object. If this property is confirmed by the compiler, there are much room for getting free methods. Employing the dynamic method replacement makes it possible for the compiler to detect this property, since we can replace the RW method on the object with an appropriate RO method if the method no longer does write access.

The dynamic method replacement also gains the opportunity to confirm the former condition for a method to be free. However, the dynamic method replacement changes the set of methods which can be invoked (or *method state* for short) dynamically; that is, the object with replaceable methods has more mutable part than others. Thus, to perform both method lookup and method invocation without mutual exclusion, it is necessary not only that the method is free but also that the method is no longer replaced. Moreover, for a method to be free, it must be confirmed that the read part will be never written by any method not only in the current method set but also in all the method sets that the object may have in the future. Therefore, we must assume that the compiler may fail to detect free methods within a limited time and space complexity. This is the reason why the non-blocking read-only methods are incorporated in the language rather than directly using free methods.

Of course, by the dynamic method replacement, we can simply replace an RW method with an RO method. A simple RO method does not eliminate bottlenecks as perfectly as a free method does; but it eliminates some bottlenecks since the RO method can be executed with less strict mutual exclusion than the RW method. For this reason, the dynamic method replacement is still effective even though the detection of free methods sometimes fails due to the limitation of the compiler.

For example, in a binary tree dictionary class (Figure 5), the `insert` method no longer updates the variables of the object once it creates both `left` and `right` child nodes. The `insert` RW method is replaced with the `insertF` RO method by the replacement statement:

```
setmethod(insert, insertF);
```

The replaced method, `insert`, is defined with a keyword `replaceable` to indicate that method is replaceable. After the dynamic method replacement of this example, since both `insert` and `search` methods are free and the dynamic method replacement is no longer performed, we should exploit the free methods. This property should be detected by the compiler, since there are only two possible method sets (i.e., the number of combinations is not so large).

The effect of the dynamic method replacement can be explained more precisely in terms of *message renaming*. For example, let us consider the following replacement statement:

```
setmethod(m1, m2);
```

The effect is that `m1` messages sent to the object will be renamed to `m2`. The renamed `m2` message is resent to the object for further method lookup. In other words, when the object `obj`'s method is replaced, the message passing "`obj.m1(...)`" is executed as if it were "`obj.m2(...)`." If the method `m2` is also replaceable, the further renaming may take place.

When the class with replaceable methods is inherited, both the replaceable method and the substitutive method can be overridden as usual. Actually, the substitutive method may be simply a usual method. The late binding mechanism still works but it is extended: the method lookup is based on not only the actual class of the object but also the current method set (or renaming rules) of the object, and the result of method lookup can be not only a method but also a message which needs further method lookup.

The replaced method and the substitutive method must have the same signature. When the methods are overloaded, the replacement is performed for each pair of methods which have the same signature.

There are two keywords, which can be specified instead of substitutive methods. To reset the replaceable method, keyword `initial` can be used as follows:

```
setmethod(m1, initial);
```

To employ the dynamic method replacement for synchronization, keyword `suspends` can be used as follows:

```
setmethod(m1, suspends);
```

This makes the thread which lookups for the message `m1` suspend its execution until the method is replaced with a normal method as was described in Section 2.2.

The update for the dynamic method replacement is performed together with the atomic update of the instance variables of the object. The mutual exclusion for the method lookup is required together with the mutual exclusion for the method invocation as was described in Section 2.3. It should be noticed that the mutual exclusion for the RO method and that for the RW method differ but which type of method will be obtained is unknown at the moment of starting method lookup.

4 Free method detection algorithm

This section outlines an algorithm for the detection and exploitation of free methods. In this section, `suspends` keyword is ignored for simplicity.

The task of the method lookup can be regarded as a function which maps messages (ranged over by a) to methods (ranged over by m). As is described in Section 5, basically, a method table (T) is prepared for each class. If free methods are exploited, however, several method tables will be prepared for that class. Every T maps messages to table entries (ranged over by e):

$$T = \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\}$$

The storage for the current state of each replaceable method is prepared in every object with that replaceable method. The location x of the storage is common to the subclasses. The table entry e is either of an actual method m with its type (RO, RW, or Free) or a location x:

$$e ::= \mathrm{RO}\, m \mid \mathrm{RW}\, m \mid \mathrm{Free}\, m \mid \mathrm{Replaceable}\, x$$

where '$\mathrm{Free}\, m$' is used only when a free method is exploited.

A method state is represented by $M = (T, S, H, R)$, where R is the renaming rule which maps the locations to the corresponding messages:

$$R = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$$

and $S(m)$ gives a set of possible method replacements (also denoted by R) by the execution of method m:

$$S = \{m_1 \mapsto \{R_{11}, \ldots, R_{1l_1}\}, \ldots, m_n \mapsto \{R_{n1}, \ldots, R_{nl_n}\}\}$$

and the set $H = \{a_1, \ldots, a_n\}$ consists of hidden messages which are introduced behind replaceable methods for the implementation. Method replacement R updates the renaming rule R_1 to R_2 (denoted by $R_1 R$) by overwriting the relevant locations as follows:

$$\begin{array}{l} \mathrm{Dom}(R_2) = \mathrm{Dom}(R_1) \cup \mathrm{Dom}(R) \\ \quad\text{and} \\ R_2(x) = \begin{cases} R_1(x) & \text{if } x \in \mathrm{Dom}(R_1) - \mathrm{Dom}(R) \\ R(x) & \text{if } x \in \mathrm{Dom}(R) \end{cases} \end{array}$$

The method lookup function *lookup* can be defined as follows:

$$\mathit{lookup}(a, T, R) = \mathit{rename}(T(a), T, R)$$

where,

$$\begin{array}{l} \mathit{rename}(\mathrm{RO}\, m, T, R) = m \\ \mathit{rename}(\mathrm{RW}\, m, T, R) = m \\ \mathit{rename}(\mathrm{Free}\, m, T, R) = m \\ \mathit{rename}(\mathrm{Replaceable}\, x, T, R) = \mathit{lookup}(R(x), T, R) \end{array}$$

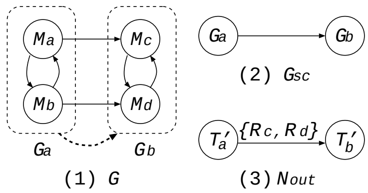

Figure 6: Graphs for free method detection algorithm

The intuitive meaning of the above definitions is pictured in Figure 7 and Figure 8.

Now, we can find a possible transition of method states $M_1 \longrightarrow M_2$ by the following definition:

$$
\begin{array}{lll}
M_1 \longrightarrow M_2 & \text{iff} & (\exists m \in MS(T_1,\, H_1,\, R_1)\,.\,\exists R \in S_1(m)\,.\,R_2 = R_1 R) \wedge \\
& & T_1 = T_2 \ \wedge\ S_1 = S_2 \ \wedge\ H_1 = H_2 \\
& & \text{where } M_1 = (T_1,\, S_1,\, H_1,\, R_1) \\
& & \text{and } M_2 = (T_2,\, S_2,\, H_2,\, R_2)
\end{array}
$$

where $MS(T,\, H,\, R)$ gives the invokable method set and is defined as follows:

$$MS(T,\, H,\, R) = \{m \mid \exists a \in (\mathrm{Dom}(T) - H)\,.\,m = lookup(a,\, T,\, R)\}$$

With the above definitions, we can outline the free method detection algorithm as follows (see Figure 6):

[**Free method detection algorithm:**]

input: an initial method state $M_0 = (T_0,\, S_0,\, H_0,\, R_0)$

output: a directed acyclic graph (DAG) with transition conditions $N_{out} = (G_{out}, c_{out})$

where

G_{out}: DAG of the adaptive method tables

c_{out}: function which maps edges of G_{out} to transition conditions $\{R_1, \ldots, R_n\}$

$\{R_1, \ldots, R_n\}$: transition condition that the renaming rule is either of $R_1, \ldots, R_n$.

The meaning of the output:

when the transition condition is satisfied, the method table can be switched along with the edge of G_{out}.

1. Let directed graph $G_0 = (V_0, E_0) = (\{M_0\}, \{\})$.

2. To generate G (the graph of all possible method states), let $G_{n+1} = (V_{n+1}, E_{n+1}) = (V_n \cup \{M' \mid \exists M \in V_n.\, M \longrightarrow M'\},\, E_n \cup \{(M, M') \mid \exists M \in V_n.\, M \longrightarrow M'\})$ repeatedly until $G = G_{n+1} = G_n$ (Figure 6–(1)).

3. Find all strongly connected components on G; let G_{sc} be the DAG of those components (Figure 6–(2)).

4. For each strongly connected component, find free methods. To save the amount of computation, the detection is performed from the child vertexes to parent vertexes. $(a,\, m)$ is a free method if (1) m is of RO type, (2) for all present and future components, the result of *lookup* by a is always m and (3) for all present and future components, there is no method (in method sets calculated by MS) which writes the instance variables read by m.

5. To prepare multiple method tables, for each strongly connected component, T is updated to $T' = T\{a_1 \mapsto \text{Free } m_1, \ldots\}$ for all found free methods.

6. To know the conditions of the table switches in advance, let G_{out} be the DAG (isomorphic to G_{sc}) of the updated T's and let $c_{out}(E) = \{R \,|\, (T\,,\, S\,,\, H\,,\, R)$ is either of terminal vertexes of the G edges which correspond to $E\}$ for edge E of G_{out}. (Figure 6–(3))

For the binary tree dictionary class (Figure 5), G in the above algorithm is as follows (ignoring `search` method for simplicity):

$$G = (\{M_0, M_1\}, \{(M_0, M_1)\})$$

where,

$$\begin{array}{ll}
M_0 = (T_0,\, S_0,\, H_0,\, R_0) \quad M_1 = (T_1,\, S_1,\, H_1,\, R_1) & \\
T_0 = T_1 = & \{\texttt{insert} \mapsto \text{Replaceable}\, x_{\text{insert}}, \texttt{insertF} \mapsto \text{RO}\, m_{\text{insertF}}, \\
& \quad a_{\text{insert}} \mapsto \text{RW}\, m_{\text{insert}}\} \\
S_0 = S_1 = & \{m_{\text{insert}} \mapsto \{\{x_{\text{insert}} \mapsto \texttt{insertF}\}\}, m_{\text{insertF}} \mapsto \{\}\} \\
H_0 = H_1 = \{a_{\text{insert}}\} & \\
R_0 = \{x_{\text{insert}} \mapsto a_{\text{insert}}\} \quad R_1 = \{x_{\text{insert}} \mapsto \texttt{insertF}\} &
\end{array}$$

In this example, each of M_0 and M_1 is a strongly connected component and finally we will obtain:

$$N_{out} = (G_{out}, c_{out}),\; G_{out} = (\{T_0, T_1'\}, \{(T_0, T_1')\}), c_{out} = \{(T_0, T_1') \mapsto \{R_1\}\}$$

where,

$$\begin{array}{ll}
T_1' = & \{\texttt{insert} \mapsto \text{Free}\, m_{\text{insertF}}, \\
& \texttt{insertF} \mapsto \text{Free}\, m_{\text{insertF}},\; a_{\text{insert}} \mapsto \text{RW}\, m_{\text{insert}}\}
\end{array}$$

Note that RW m_{insert} is no longer invoked since $a_{\text{insert}} \in H_1$ holds.

Since the space complexity of the given algorithm is exponential to the number of replaceable methods, the compiler may fail to detect free methods within limited computational resources. In such cases, the compiler will abort the detection and solely use the original method table.

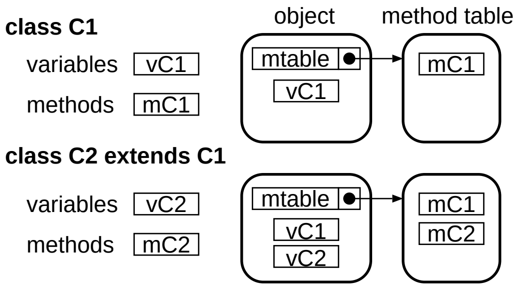

Figure 7: The representation of object

5 Implementation for a shared memory architecture

Our target machine is the SGI's shared-memory computer, POWER Onyx (12 MIPS R10000 processors, 195MHz, 2GB memory). The OPA compiler generates C code, then the C compiler generates the binary code. The OPA runtime initially creates an OS thread (a process sharing the memory with other processes) on each processor. The OPA runtime non-preemptively schedules a number of OPA threads on the top of the fixed number of OS threads which will run actually in parallel on the same number of processors.

5.1 Implementation of mutual exclusion

The RW-RW mutex is implemented with (a) a flag which indicates whether or not RW section is being executed by a thread and (b) a thread queue to which threads waiting for their turn are enqueued. To modify the flag and the queue atomically, lock/unlock operations are employed. In the actual implementation, the lock operation is deferred until the first method lookup is finished, and not performed if the obtained method is of RO type. The detail of the timing of the lazy lock is described in Section 5.3.

The RO-W mutex (to prohibit the RO method from reading the state which is partly updated) is implemented with a version numbering method: every object maintains a version number (initially 1), which is added by 2 with every update of the object and is set to 0 during the update. The reader retries if the version number is 0 (i.e., it is during the update) or it changes after reading. This method is a variant of Lamport's version numbering method[LAM 77] that requires two version counters. The benefit of this method is that it permits concurrent reads and does not require costly lock operations.

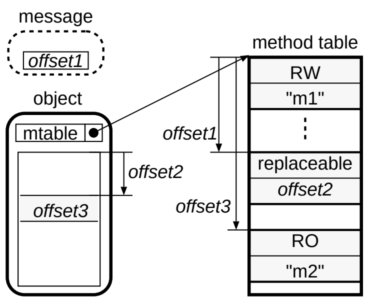

Figure 8: The method lookup for replaceable method

5.2 Representation of object

OPA objects are implemented with C structures composed of variables for mutual exclusion, a pointer to the method table for that class, and an instance variable table. Objects of its subclass are implemented with similar C structures which may have appended methods and instance variables; but they keep, as usual, the shape of its superclass to realize subtyping with little overhead (see Figure 7). The status of method replacement is stored in the instance variable table.

Each entry of the method table for a class usually holds the method information that consists of a method type (`RO` or `RW`) and a function pointer for the method (see Figure 8). However, if the method is replaceable, the method information consists of the method type `Replaceable` and an offset for the replacement status in the instance variable table. The replacement status consists of solely an offset in the method table for the substitutive method.

5.3 The lookup and invocation of replaceable methods

Figure 9 illustrates the pseudo code of the method lookup and the method invocation for a message passing expression of `o.message(args)`.

The thread first confirms that the object is not being updated (i.e., the version number `o.vn` is not zero) and memories the current version number in `vn`. Then the thread fetches a method table and lookups the method with an offset (*offset1* in Figure 8) given in the message. If the method is not replaceable, the thread will get the reference to the method code. However, if the method is replaceable, the entry holds an offset (*offset2*) for the replacement status in the instance variable table. This replacement status contains a new offset (*offset3*) for further method lookup. This process is repeated until a non-replaceable method is obtained.

If the obtained method is of RO type, the RO method can be invoked without acquiring the lock unless the update of the object is detected until

```
START:
do { vn=o.vn; } while(vn==0);
RETRY:
msg = message;
LOOKUP:
e = lookup(o, msg); method = e.method;
switch(e.mtype) {
case Replaceable:
  i = method;  msg = o[i]; goto LOOKUP;
case RO:
  if (vn && vn != o.vn) goto ROfail;
  status = invoke(method, vn, o, args);
  if (status == ROfail) {
ROfail:
    lock(o); vn=0; goto RETRY;
  }
  break;
case RW:
  if (vn) {
    lock(o);
    if (vn != o.vn){vn=0; goto RETRY;}
  }
  if (o.use) { suspends(o); goto START; };
  status = invoke(method, o, args);
  break;
case Free:
  if (vn == 0) unlock(o);
      /* 0 means that lock has been acquired */
  status = invoke(method, 2, o, args);
      /* 2 means that the method is free */
  break;
}
```

Figure 9: Pseudo code of method lookup and method invocation for o.message(args) on shared memory architectures

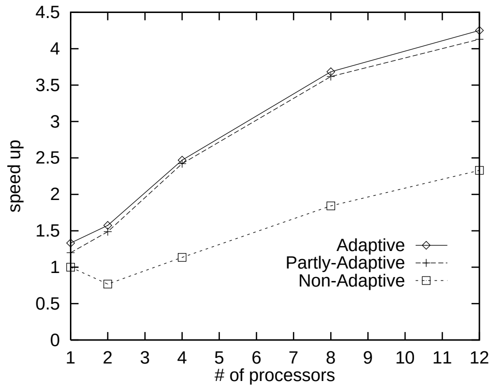

Figure 10: The speedup with dynamic method replacement (Adaptive/Partly-Adaptive) on a shared memory architecture (relative speedup to Non-Adaptive 1 PE case: the elapsed time is 0.464s for 30,000 nodes insertion to a shared binary tree.)

the RO method finished the atomic read of instance variables. If an update is detected, the thread retries the same sequence from the method lookup but with the lock (`vn`=0 indicates this) to avoid further retries.

On the other hand, if the obtained method is of RW type, the thread locks the object and checks the use flag. If the object is in use by some other execution of RW section, the thread enqueues itself into the thread queue of the object and suspends its execution. If the object is not in use, the thread invokes the method which will set the use flag and release the lock.

6 Implementation for a distributed memory architecture

For a distributed memory architecture, we use a workstation cluster which consists of 4 Sun Microsystems SPARCstation5 (CPU: 110MHz microSPARC-II) workstations perfectly connected with each other by high speed serial communication interfaces called STAFF-link[NIK 97], which exhibits approximately 70$\sim$80μs latency for a small size ($\leq$ 64bytes) message.

The OPA runtime initially creates a process on each workstation. The OPA runtime non-preemptively schedules a number of OPA threads on those processes which will run actually in parallel on the same number of processors.

Since the objects on a remote node is not directly accessible, either a remote method invocation to the object or a remote fetch of the object is required for the message passing to the object. We currently employ the former method.

The latency for the access to a remote object must be eliminated or hidden. The elimination of remote access can be realized by using free methods: the

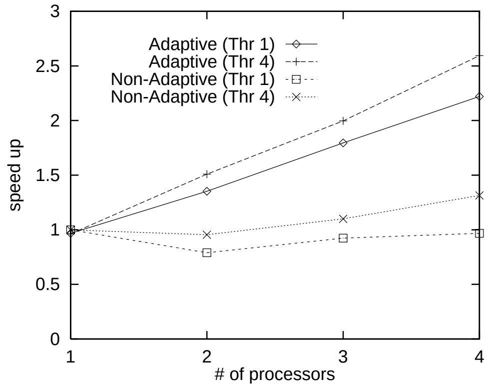

Figure 11: The speedup with dynamic method replacement (Adaptive) on a distributed memory architecture (relative speedup to Non-Adaptive 1 PE case: the elapsed time is 2.80s for 30,000 nodes insertion to a shared binary tree.)

invocation of the free method can be performed on the local copy (proxy) of the remote object. The latency hiding is realized by using multiple threads on each processor. The OPA runtime manages the OPA (language-level) threads non-preemptively and very efficiently by performing the context switch fully by the compiled code.

The proxy is created when the reference to the corresponding remote object is received from the network for the first time. The proxy is created with the same size as the corresponding actual object. After that, the reference to the remote object, when it is received from the network, is converted into the reference to the proxy.

When the proxy's method is invoked, the proxy basically performs remote message passing to the remote actual object via a server on the remote node. When the server returns the return value to the proxy, the server also checks if the invoked method has been replaced with a free method: if it is a free method, the server attaches the data (method table and fields) of the actual object to the return value message. Once the proxy has the free method and the immutable copy of the fields, the subsequent method invocations will be performed entirely locally.

7 Performance measurements on parallel architectures

The performance measurements are performed with programs exploiting the binary tree class in Figure 5. We compared the following three programs. (Each program inserts 30,000 nodes with random keys into the shared tree structure):

Adaptive the program that exploits the dynamic method replacement and the resultant free methods (i.e., after dynamic method replacement, the `insert` method becomes free.),

Partly-Adaptive the program that exploits the dynamic method replacement but does not involve free methods (i.e., after dynamic method replacement, the `insert` method simply becomes of RO type.),

Non-Adaptive the program that has no replaceable methods.

Figure 10 shows the relative speedups to Non-Adaptive 1 PE case (the elapsed time is 0.464s.) for three programs on the shared memory architecture. The result shows the programs exploiting the dynamic method replacement is faster than the non-adaptive program for all 1 to 12 PEs. Since the speedup is achieved, we can say the bottlenecks are eliminated to some degree with the dynamic method replacement. The difference of Partly-Adaptive from Adaptive is not significant except for 1 PE case, because copying of the immutable parts of objects (which is achieved by software in the distributed memory architecture) is automatically performed by hardware cache.

For the distributed memory parallel architecture, each program creates 1 or 4 thread(s) on each processor (for the latency hiding). Figure 11 shows the relative speedups to Non-Adaptive 1 PE case (the elapsed time is 2.80s.) for Adaptive and Non-Adaptive programs. The result shows the programs exploiting the dynamic method replacement is approximately 2 times faster than the non-adaptive program for 2 to 4 PEs. The result also indicates the latency hiding is effective on distributed memory architectures.

8 Related work

Actors[AGH 87] has a language construct `become`, which dynamically change the behavior of objects. This mechanism corresponds to a mechanism which switches the entire classes dynamically. Although OPA compiler exploits the switch of the entire method table for free methods, it is too low level to be programmed directly.

Concurrent Aggregates[CHI 93] introduces the notion of aggregate, a set of the same kind of objects, called representatives. When a message is sent to an aggregate, the message is sent to an arbitrarily chosen representative. Then, if necessary, the representative forwards the message to an appropriate object. This mechanism also eliminates bottlenecks around frequently accessed objects. OPA provides the same kind of object (such as root node of `BinTree` class) via more systematically confirmed way. Again, We think it is a little bit low level to program the aggregate directly.

9 Conclusion

In this paper, we proposed *dynamic (runtime) method replacement* which provides adaptive objects for concurrent accesses, where we can increase the number of read-only methods and the immutable part of the objects dynamically to eliminate or relax the mutual exclusion and remote communication. This paper also presented its implementation and the results of the performance measurements on parallel architectures, which indicate the effectiveness of our approach.

The proposed feature was implemented and examined in our object-oriented parallel programming language OPA. Future work includes performance measurements using practical applications; we have already tested the N-body problem application using tree structure and observed a considerable bottleneck elimination.

References

[AGH 87] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[AG 96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley Publishing Company, 1996.

[BFC 95] P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor Classification. *ACM Computing Surveys*, Vol. 27, No. 1, pp.63–72, 1995.

[CHI 93] A. A. Chien. *Concurrent Aggregates.* The MIT Press, 1993.

[LAM 77] L. Lamport. Concurrent Reading and Writing. *Communication of the ACM*, Vol. 20, No. 11, pp. 806–811, 1977.

[NIK 97] H. Nakajo, A. Ichikawa, and Y. Kaneda. An Implementation and Evaluation of a Distributed Shared-Memory System on Workstation Clusters Using Fast Serial Links. In *Proc. of Int.Symp. on High Performance Computing(ISHPC)*, Vol. 1336 of *LNCS*, pp. 143–158. 1997.

[SAR 93] J. Sargeant. United Functions and Objects: an Overview. Technical Report, Department of Computer Science, University of Manchester, 1993.

[TMY 94] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f : A Future-based Polymorphic Typed Concurrent Object-Oriented Language —its Design and Implementation—. *DIMACS workshop on Specification of Parallel Algorithms*, No. 18, pp.275–292, 1994.

[TY 95] K. Taura and A. Yonezawa. Schematic: A Concurrent Object-Oriented Extension to Scheme. Technical Report, Department of Information Science, University of Tokyo, 1995.

[YT 96] M. Yasugi and K. Taki. OPA: An Object-Oriented Language for Parallel Processing — its Design and Implementation —. *IPSJ SIG Notes 96-PRO-8(SWoPP'96)*, Vol. 96, No. 82, pp. 157–162, August 1996. (in Japanese).