A Table-Driven Approach of Garbage Collection for Active Objects[☆☆]

Chung Yung, Wen-Kai Tu, and Cheng-Chun Hsu
Department of Computer Science and Information Engineering
National Dong Hwa University, Hualien, Taiwan 97401
yung@mail.ndhu.edu.tw, m9321037@em93.ndhu.edu.tw, and m9721062@ems.ndhu.edu.tw

Abstract

This paper proposes a Table-Driven approach of garbage collection for active objects. In the actor model, a software system consists of autonomous computational agents called actors that encapsulate data as well as some primitive processing power to manipulate data. Garbage collection is an important service of automatic storage management. Based on the garbage of actors defined by Nelson, we propose a Table-Driven approach to reclaim the unused objects in the active object systems. We define two auxiliary tables to keep tracks of the actors with the same colors. Improved from the Push-Pull algorithm and the Is-Black algorithm for garbage collection proposed by Kafura et al., our algorithm reduces the redundant repeated visits to actors in the marking phase. The time complexity of our Table-Driven algorithm is $O(N^2)$, which is the same as the Push-Pull and Is-Black algorithms. The experiments in our simulation show the effectiveness and efficiency of our new algorithm. The execution time of our table-driven garbage collector is reduced by 2.128% to 25.129% over the Is-Black algorithm; and by 33.191% to 64.980% over the Push-Pull algorithm.

1. Introduction

This paper proposes a new approach of garbage collection for active objects. Garbage collection is an important service of automatic storage management, as it simplifies the programming model by saving the programmer time while avoiding bugs and memory leaks [**?** **?**]. In the actor model, a software system consists of autonomous computational agents called *actors* that encapsulate data as well as

[☆]This work is partially supported by the National Science Council under grant no. NSC 97-2218-E-259-001.

some primitive processing power to manipulate data [**?** **?**]. The focus of this paper is to develop a new garbage collection approach for the actor model that can be applied to the embedded systems.

In conventional object-oriented systems, the problem of garbage collection is usually expressed as a problem of abstract graph [**?** **?** **?**]. A directed graph, called the *object reference graph*, is used to represent an object as a node, and an object reference as an edge. A subset of the objects which are always considered as non-garbage are called *root* objects. Thus, the garbage collection problem of the object-oriented systems is reclaiming all the objects which cannot affect the user computation.

In the actor model, actors extend sequential objects by encapsulating a thread of control along with procedures and data in the same entity, and thus actors provide a unit of abstraction and distribution in concurrency [**?**]. Actors are inherently concurrent and autonomous enabling efficiency in parallel execution [**?**] and facilitating mobility [**?**]. The actor model provides a useful framework for understanding and developing object-oriented systems in open distributed environments [**?** **?** **?** **?**].

The problem of garbage collection of actors is slightly different from garbage collection for the conventional objects, and it needs that special algorithms to be devised for garbage collecting actors [**?** **?** **?** **?** **?**].

- In conventional object-oriented systems, the model of computation is independent of threads of control manipulating the objects they can refer to. Implicit in this idea is that the threads of control by themselves are always important. Thus, an important subset of the root set consists of the objects referenced from the stacks of running threads [**?**].

- In actor systems, encapsulating a thread of control within each object creates a uniform object model which combines the distinct notion of object mobility and process migration [**?**].

- In actor systems, the thread of control is encapsulated in the object itself. Conceptually, there are no stacks, registers or a separate heap. The idea is that no actor would otherwise ever become garbage because each actor is referenced by the thread of control it encapsulates [**?**].

Nelson formulated certain coloring rules to identify actor garbage [**?**]. Basically, a garbage actor is one which is:

1. not a root actor, and
2. cannot potentially receive a message from a root actor, and
3. cannot potentially send a message to a root actor.

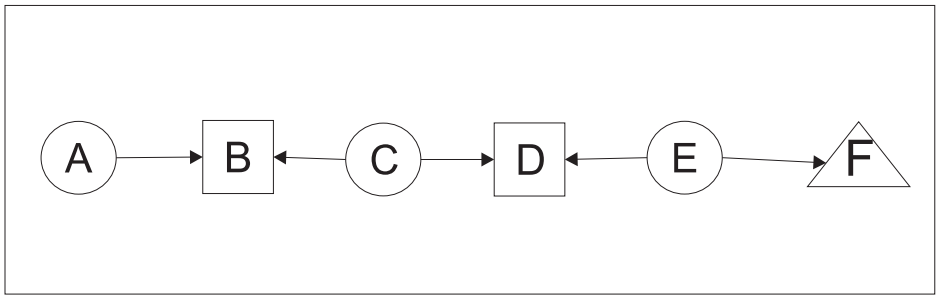


Figure 1: An Example of the Actor Reference Graph.

Based on these rules, Kafura et al. present two algorithms for actor garbage collection; namely, the Push-Pull algorithm and the Is-Black algorithm [**?**].

- In the Push-Pull algorithm, there are two coroutines, a Pusher and a Puller, to move actors between black, gray and white sets. The Pusher operates on the white set and tries to push a white actor into the gray or black set depending on whether it has an acquaintance which is black or gray. The Puller pulls acquaintances of black actors out of the gray or white set into the black set.

- In the algorithm Is-Black, one rule repeatedly colors black the acquaintances of black actors. A second rule does a depth first search from all active actors for a black actor. If a black actor is found, then the originating actor is colored black.

Both these algorithms have a space complexity of O(N) and time complexity of O(N^2) where N is the number of actors in the system [**? ?**].

Figure **??** is an example of the actor reference graph. Actor F is the root actor, actors A, C, and E are processing actors, and actors B and D are blocked actor. We apply the Push-Pull algorithm and the Is-Black algorithm to Figure **??**. We observe that both algorithms repeatively visit some of the actors, and this causes inefficiency.

In this case of applying the Is-Black algorithm, for example, the first iteration of the Is-Black algorithm visits the actors A, B, C, D, and E, and then determines that the actors E and D are non-garbage. In the second iteration, the actors A, B, and C will be visited again, then it is determined that the actors C and B are non-garbage. In third iteration, the actor A will be visited again, then it is determined that the actor A is non-garbage. Note that, in this example, the actor A is visited three times, the actors B and C are visited twice. According to our observation, these repeated visits are redundant and can be eliminated.

The goal of this paper is to develop a new garbage collection approach which improves the efficiency by eliminating the redundant repeated visits in the Push-Pull algorithm and the Is-Black algorithm.

This paper is organized as follows. Section 2 is a brief description of the Push-Pull algorithm and the Is-Black algorithm. We present our Table-Driven algorithm in Section 3. We analyze the correctness and the complexity of our Table-Driven approach in Section 4. Section 5 shows our experiments that compare our Table-Driven algorithm with the Push-Pull algorithm and the Is-Black algorithm. At last is a brief conclusion.

2. Related Work

A formal definition of garbage actors is given by Kafura et al. [**? ?**].

Definition (Garbage actors): A garbage actor is one which:

- is not a root actor, or
- cannot potentially receive a message a message from a root actor, and
- cannot potentially send a message to a root actor. □

Based on the definition, they mark actors with three colors:

- White: The actors are not reachable from a root actor.
- Gray: The actors are reachable from a root actor, but can not become processing.
- Black: The actors are non-garbage. They are either root actors, or are both reachable from a root actor and potentially processing.

According to the coloring rules, Kafura et al. present two algorithms for actor garbage collection; namely, the Push-Pull algorithm and the Is-Black algorithm. In this section, we describe the both algorithms in more details.

2.1. The Push-Pull Algorithm

Kafura et al. propose the Push-Pull algorithm to implement the coloring rules using two coroutines, a Pusher and a Puller, to move actors between black, gray and white sets.

- The Pusher operates on actors in the white set and pushes actors from the white set into the gray or black set, depending on whether it has an acquaintance which is black or gray.

- The Puller operates on actors in the black set and pulls actors from the white and gray sets into the black set.

The actions of the Push-Pull algorithm are illustrated using the actor system shown in Figure **??**. For simplicity, we assume that actors are examined in alphabetical order.

1. The initialization step puts F in the black set and all other actors in the white set.
2. In the first pass, the Pusher puts E into the black set since it has an acquaintance F. The Puller puts D into the black set since it is an acquaintance of E.
3. In the second pass, the Pusher puts C into the black set since it has an acquaintance D. The Puller puts B into the black set since it is an acquaintance of C.
4. In the third pass, the Pusher puts A into the black set since it has an acquaintance B. The Puller takes no actions and it terminates.

At termination, the black set contains actors A, B, C, D, E, and F. They are non-garbage.

2.2. The Is-Black Algorithm

The Is-Black algorithm uses two coloring rules and a *visit* field:

- The first coloring rule in the loop colors black acquaintances of black actors.
- The second coloring rule does a depth first search from active actors for a black actors. If a black actor is found, the actor initiating the search is colored black.
- The *visit* field is used to detect cycles during the depth first search.

The actions of applying the Push-Pull algorithm to the actor system of Figure **??** is presented in Section 1.

3. A Table-Driven Approach of Garbage Collection

In this section, we present a new Table-Driven approach of garbage collection for active objects. Since both the Push-Pull algorithm and the Is-Black algorithm induce redundant repetitive visits to actors, we propose using two auxiliary tables to trace the reachability of actors efficiently.

Section 3.1 gives the definitions and notations used in our algorithm. We introduce the Table-Driven algorithm in Section 3.2. Finally, we give a complete example of applying the Table-Driven algorithm.

3.1. Definitions and Notations

In this section, we define the notations of actor systems used in our algorithm.

In the actor model, a software system consists of autonomous computational agents called *actors* that encapsulate data as well as some primitive processing power to manipulate data. We divide the actors into three categories:

- **Root actors**: Root actors are the actors which directly communicate with the external environment and are always considered non-garbage.
- **Processing actors**: Processing actor are the actors which are either processing a message or have messages pending in their mail queues.
- **Blocked actors**: Blocked actors are the actors which are not processing and not root actors.

The symbols for the three categories of actors are shown in the following table.

Symbol	Meaning
△	Root actors
◯	Processing actors
□	Blocked actors

For simplicity, our Table-Driven algorithm implements the definition of a *non-garbage* actor as follows.

> Definition (Non-garbage actors): A non-garbage actor is the actor which
>
> - is a root actor, or
> - can potentially receive a message a message from a root actor, or
> - can potentially send a message to a root actor. □

In this definition, the term "potentially" requires further clarification [**?**]. An actor can send a message to a root actor only if the actor is processing and has the root actor as a direct acquaintance. While the execution of an actor system, there is a set of transformations that can change an actor reference graph from a current representation into what can potentially happen. There are two transformations concerning changes in the states of actors and changes in the topology of the system.

- First, sending a message from a processing or root actor to a blocked acquaintance allows the blocked actor to become processing. This transformation reflects the ability of an actor to alter the state of the other actors.

- The second transformation occurs when a root or processing actor sends the mail queue addresses of its own or its acquaintances to the other actors. This transformation reflects the ability of an actor to change the topology of the system.

One key property of garbage actors is that once they are recognized as garbage they cannot become non-garbage afterwards. This is because actors are determined to be garbage only when there is no possibility of communication between it and a root actor.

Our Table-Driven algorithm marks the actors with two colors:

- **Black**: When an actor is determined as non-garbage, it is colored black.
- **White**: If an actor is considered as garbage, it is colored white.

3.2. The Table-Driven Algorithm

In this section, we present the Table-Driven algorithm that we propose for garbage collection of active objects.

An actor reference graph $\mathcal{G}$ is expressed by a quadruple $\langle \mathcal{R}, \mathcal{P}, \mathcal{B}, \mathcal{E} \rangle$, where $\mathcal{R}$ is the set of root actors, $\mathcal{P}$ is the set of processing actors, $\mathcal{B}$ is the set of blocked actors, and $\mathcal{E}$ is the set of reference edges. A reference edge $e \in \mathcal{E}$ is denoted by $\langle u, v \rangle$, where $u, v \in \mathcal{R} \cup \mathcal{P} \cup \mathcal{B}$. We use the notation $\mathcal{E}[u]$ to denote the set $\{v \in \mathcal{R} \cup \mathcal{P} \cup \mathcal{B} \mid \langle u, v \rangle \in \mathcal{E}\}$.

According to our observation, both the Push-Pull algorithm and the Is-Black algorithm induce redundant repeated visits to actors. In the Table-Driven algorithm, we use two auxiliary tables to eliminate the redundant visits; namely, the *processing ancestors table* $\mathcal{A}$ and the *processing friends table* $\mathcal{F}$.

The processing ancestors table $\mathcal{A}$ is defined as follows.

> Definition (Processing ancestors table) $\mathcal{A}$: For $u \in \mathcal{B}$ and $v \in \mathcal{R} \cup \mathcal{P}$, $\langle u, v \rangle \in \mathcal{A}$ if
>
> - $\langle v, u \rangle \in \mathcal{E}$, or
> - $\exists u_1, \ldots, u_k \in \mathcal{B}$ such that $\langle v, u_1 \rangle$, $\langle u_1, u_2 \rangle$, $\ldots$, $\langle u_{k-1}, u_k \rangle$, and $\langle u_k, u \rangle \in \mathcal{E}$. □

For $u \in \mathcal{B}$, we write $\mathcal{A}[u] = \{v \in \mathcal{R} \cup \mathcal{P} \mid \langle u, v \rangle \in \mathcal{A}\}$.

The processing friends table $\mathcal{F}$ is defined as follows.

> Definition (Processing friends table) $\mathcal{F}$: For $u, v \in \mathcal{R} \cup \mathcal{P}$, $\langle u, v \rangle$ and $\langle v, u \rangle \in \mathcal{F}$ if

Algorithm: *Table-Driven*

TD_Initialization;

$Q := \mathcal{R}$;
WHILE EXISTS [$a \in Q$] DO
  a.color := *black*;
  a.visit := *true*;
  IF [*a* is blocked]
    $Q := Q \cup \{x \in \mathcal{E}[a] \cup \mathcal{A}[a] \mid x.visit = false\}$;
  ELSE // *a* is root or processing
    $Q := Q \cup \{x \in \mathcal{E}[a] \cup \mathcal{F}[a] \mid x.visit = false\}$;
  END IF;
END DO;

Figure 2: The Table-Driven Algorithm

- $\langle u, v\rangle \in \mathcal{E}$, or
- $\exists w \in \mathcal{B}$ such that $u, v \in \mathcal{A}[w]$,
- $\exists w \in \mathcal{B}$ such that $u \in \mathcal{A}[w]$ and $\langle w, v\rangle \in \mathcal{E}$. □

For $u \in \mathcal{R} \cup \mathcal{P}$, we write $\mathcal{F}[u] = \{v \in \mathcal{R} \cup \mathcal{P} \mid \langle u, v\rangle \in \mathcal{F}\}$.

For each actor, we use a *visit* flag to keep record whether the actor has been visited. A initialization step in the Table-Driven algorithm clears the *visit* flag of each actor, marks the *color* field of each actor as white, and constructs the auxiliary tables $\mathcal{A}$ and $\mathcal{F}$.

The Table-Driven algorithm is shown in Figure **??**. The algorithm starts with $Q = \mathcal{R}$ to trace through the actor reference graph $\mathcal{G}$ for the actors that can potentially send or receive message from a root, according the auxiliary tables $\mathcal{A}$ and $\mathcal{F}$. At termination of the Table-Driven algorithm, all the non-garbage actors are marked as black. In other words, all the white actors are garbage and can be reclaimed.

3.3. An Example

In this section, we use the example shown in Figure **??** to demonstrate how the Table-Driven algorithm finds the garbage actors.

After the initialization step, $\mathcal{A} = \{\langle C, A\rangle, \langle D, A\rangle, \langle D, G\rangle, \langle F, E\rangle\}$, and $\mathcal{F} = \{\langle A, E\rangle, \langle E, A\rangle, \langle A, G\rangle, \langle G, A\rangle\}$. The Table-Driven algorithm starts with $Q = \{A\}$.

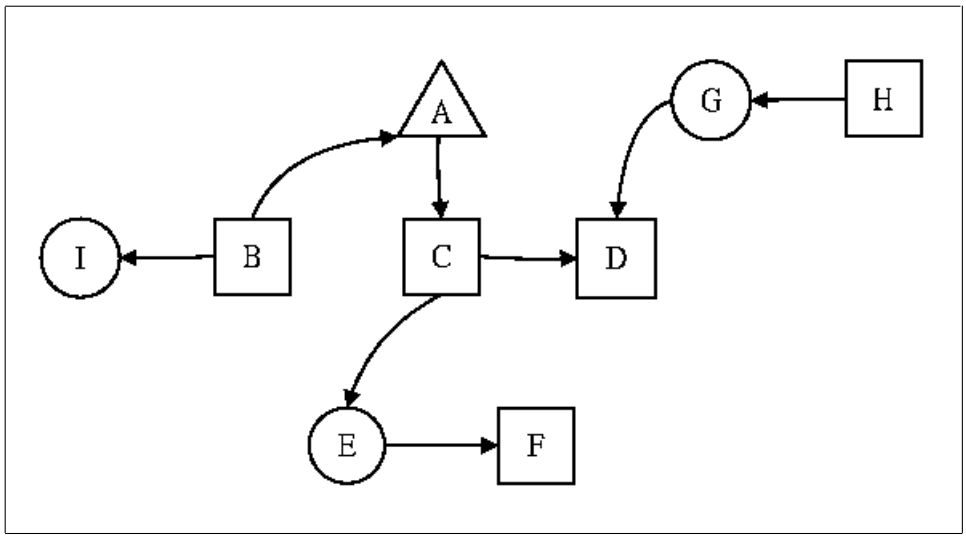


Figure 3: An example of the actor reference graph.

Next, the algorithm iterates on the elements in Q.

- Iterating on $a = A$, the algorithm puts C, E, and G into Q, and marks A black.
- Iterating on $a = C$, the algorithm puts D and E into Q, and marks C black.
- Iterating on $a = E$, the algorithm puts F into Q, and marks E black.
- Iterating on $a = G$, the algorithm puts D into Q, and marks G black.
- Iterating on $a = D$, the algorithm marks D black.
- Iterating on $a = F$, the algorithm marks F black.

At termination, the actors A, C, E, G, D, and F are marked black. Hence, the actors B, H, and I are garbage and may be reclaimed.

4. Analysis of the Table-Driven Algorithm

In this section, we analyze the correctness and computational complexity of our new algorithm.

4.1. Correctness

For proving the correctness of the Table-Driven algorithm, we need a few lemmas.

Lemma 1: If $\exists\langle u, v\rangle \in \mathcal{E}$ and $u \in \mathcal{P} \cup \mathcal{R}$, then

- if u is black then v is black, and
- if v is black then u is black.

In other words, u and v are of the same color.

Proof:

1. If u is black, then either u potentially sends messages to a root, or u potentially receives messages from a root. Since $\langle u, v\rangle \in \mathcal{E}$, v can potentially send messages to a root, or can potentially receive messages from a root. And hence, v is black.
2. If v is black, then either v potentially sends messages to a root, or v potentially receives messages from a root. Since $\langle u, v\rangle \in \mathcal{E}$ and $u \in \mathcal{P} \cup \mathcal{R}$, u can potentially send messages to a root, or u can potentially receive messages from a root. And hence, u is black.

By 1 and 2, we complete the proof. □

Note that Lemma 1 is true no matter v is a root, processing or blocked actor. Next, we use Lemmas 2 and 3 to show the reasons of using auxiliary tables $\mathcal{A}$ and $\mathcal{F}$.

Lemma 2: If $\langle u, v\rangle \in \mathcal{A}$, then

- if u is black then v is black, and
- if v is black then u is black.

In other words, u and v are of the same color.

Proof: By definition, if $\langle u, v\rangle \in \mathcal{A}$, then $u \in \mathcal{B}$, and $v \in \mathcal{R} \cup \mathcal{P}$, and either

1. $\langle v, u\rangle \in \mathcal{E}$, or
2. $\exists u_1, \ldots, u_k \in \mathcal{B}$ such that $\langle v, u_1\rangle$, $\langle u_1, u_2\rangle$, …, $\langle u_{k-1}, u_k\rangle$, and $\langle u_k, u\rangle \in \mathcal{E}$.

Hence,

1. If $\langle v, u\rangle \in \mathcal{E}$, it applies to Lemma 1.
2. If $\exists u_1, \ldots, u_k \in \mathcal{B}$ such that $\langle v, u_1\rangle$, $\langle u_1, u_2\rangle$, ..., $\langle u_{k-1}, u_k\rangle$, and $\langle u_k, u\rangle \in \mathcal{E}$, and $v \in \mathcal{P} \cup \mathcal{R}$, u can potentially send messages to a root, or u can potentially receive messages from a root. And hence, u is black.

By 1 and 2, we complete the proof. □

Lemma 3: $\langle u, v\rangle \in \mathcal{F}$ then

- if u is black then v is black, and
- if v is black then u is black.

In other words, u and v are of the same color.

Proof: By definition, if $\langle u, v\rangle \in \mathcal{F}$, then $u, v \in \mathcal{R} \cup \mathcal{P}$, and either

1. $\langle u, v\rangle \in \mathcal{E}$, or
2. $\exists w \in \mathcal{B}$ such that $u, v \in \mathcal{A}[w]$,
3. $\exists w \in \mathcal{B}$ such that $u \in \mathcal{A}[w]$ and $\langle w, v\rangle \in \mathcal{E}$. □

Hence,

1. If $\langle u, v\rangle \in \mathcal{E}$, it applies to Lemma 1.
2. If $\exists w \in \mathcal{B}$ such that $u, v \in \mathcal{A}[w]$, u and w are of the same color by Lemma 2. And, v and w are of the same color by Lemma 2. Therefore, u and v are of the same color.
3. If $\exists w \in \mathcal{B}$ such that $u \in \mathcal{A}[w]$ and $\langle w, v\rangle \in \mathcal{E}$, u and w are of the same color by Lemma 2. And, w and v are of the same color. Therefore, u and v are of the same color.

By 1, 2, and 3, we complete the proof. □

Lemma 4: All the non-garbage actors are colored black at termination of the Table-Driven algorithm.

Proof (sketch): If u is a non-garbage actor, u potentially sends messages to a root or receives messages form a root. Hence,

1. If $u \in \mathcal{P}$, there exists a sequence of actors $u_1, u_2, \ldots, u_k \in \mathcal{P} \cup \mathcal{R}$, and $v \in \mathcal{R}$, such that $\langle u, u_1\rangle, \langle u_1, u_2\rangle, \ldots, \langle u_{k-1}, u_k\rangle$, and $\langle u_k, v\rangle \in \mathcal{F}$. Therefore, u is colored black at termination of the Table-Driven algorithm.

2. If $u \in \mathcal{B}$, there exists an actors $v \in \mathcal{P} \cup \mathcal{R}$, such that $\langle u, v \rangle \in \mathcal{A}$ and v is colored black at termination of the Table-Driven algorithm.

By 1 and 2, we complete the proof. □

Theorem: All the white actors at termination of the Table-Driven algorithm are garbage.

Proof: It is clear that the theorem is an inverse of Lemma 4. □

4.2. Space Complexity

In this section, we analyze the total space needed in the Table-Driven algorithm, which is the space needed for the two auxiliary tables $\mathcal{A}$ and $\mathcal{F}$.

For the worst case, we consider the case of an actor system consists of N root or processing actors, and N blocked actors with each actor referring to all the other actors. It is clear that

- There are $O(N^2)$ entries in table $\mathcal{A}$, and
- There are $O(N^2)$ entries in table $\mathcal{F}$.

Therefore, the space complexity of the Table-Driven algorithm is $O(N^2)$.

4.3. Time Complexity

In this section, we analyze the time complexity of the Table-Driven algorithm. The following analysis is based on the worst-case analysis.

1. Time for constructing the table $\mathcal{A}$ ($\mathcal{T}_1$): It takes $O(N^2)$ time to construct $\mathcal{A}$; that is, $\mathcal{T}_1 = O(N^2)$.
2. Time for constructing the table $\mathcal{F}$ ($\mathcal{T}_2$): It takes $O(N^2)$ time to construct $\mathcal{F}$; that is, $\mathcal{T}_2 = O(N^2)$.
3. Time for the while loop in the algorithm ($\mathcal{T}_3$): There are at most $O(N)$ actors that may be put into $\mathcal{Q}$, and it takes $O(N)$ time to compute $\mathcal{Q} := \mathcal{Q} \cup \{x \in \mathcal{E}[a] \cup \mathcal{A}[a] \mid x.visit = false\}$ or $\mathcal{Q} := \mathcal{Q} \cup \{x \in \mathcal{E}[a] \cup \mathcal{F}[a] \mid x.visit = false\}$; that is, $\mathcal{T}_3 = O(N^2)$.

So we know that the time complexity of the Table-Driven algorithm is $O(N^2)$.

Test Case	# of Actors	# of Root	# of Processing	# of Blocked	# of edges
A	302	2	140	160	538
B	505	5	200	300	539
C	1160	10	400	750	1950
D	2215	15	750	1450	3780
E	4030	30	1200	2800	6190
F	5040	40	2000	3000	5560
G	6060	60	2500	3500	6700
H	8070	70	3000	5000	8880
I	9080	80	4000	5000	10174
J	10000	90	4010	5900	11982

Figure 4: The Data of the test cases

5. Experiments

This section describes our experiments on the Table-Driven garbage collection for actors, compared with the Push-Pull algorithm and the Is-Black algorithm.

For the experiments, we implement an actor system simulator, called *GCofActorsSimulator*, which is written in *Java*. GCofActorsSimulator provides a robust and flexible environment for simulating applications of actor systems. Three garbage collection algorithms are implemented in GCofActosSimulator:

- the *Push-Pull* algorithm,
- the *Is-Black* algorithm, and
- the *Table-Driven* algorithm.

When running at GCofActorsSimulator, we may assign the number of total actors, the number of root actors, the number of processing actors, and the number of references that we would like to simulate. GCofActorsSimulator randomly generates an actor system of the assigned configuration, and simulates the execution of the generated actor system three times; with garbage collectors of the Push-Pull algorithm, the Is-Black algorithm, and the Table-Driven algorithm, respectively.

We measure the simulation of the garbage collectors on a platform with the following specification:

- ASUS desktop personal computer (model number: AB-P2300),
- A Pentium-4 CPU running with a 3GHz clock rate,

Test Case	Table-Driven	Push-Pull		Is-Black	
A	0.92ms	1.6ms	42.500%	0.94ms	2.128%
B	1.57ms	3.44ms	54.360%	1.9ms	17.368%
C	5.19ms	14.82ms	64.980%	5.81ms	10.671%
D	15.62ms	23.38ms	33.191%	16.24ms	3.818%
E	39ms	64.48ms	39.516%	50.62ms	22.955%
F	37.6ms	92.86ms	59.509%	50.22ms	25.129%
G	51.32ms	132.48ms	61.262%	59.4ms	13.603%
H	73.44ms	186.24ms	60.567%	91.94ms	20.122%
I	109.26ms	188ms	41.883%	118.38ms	7.704%
J	149.64ms	255.66ms	41.469%	158.68ms	5.697%

Figure 5: The experiment result of the test cases

- 1GB DDR RAM, and
- Running Windows XP 2002 Professional operating system.

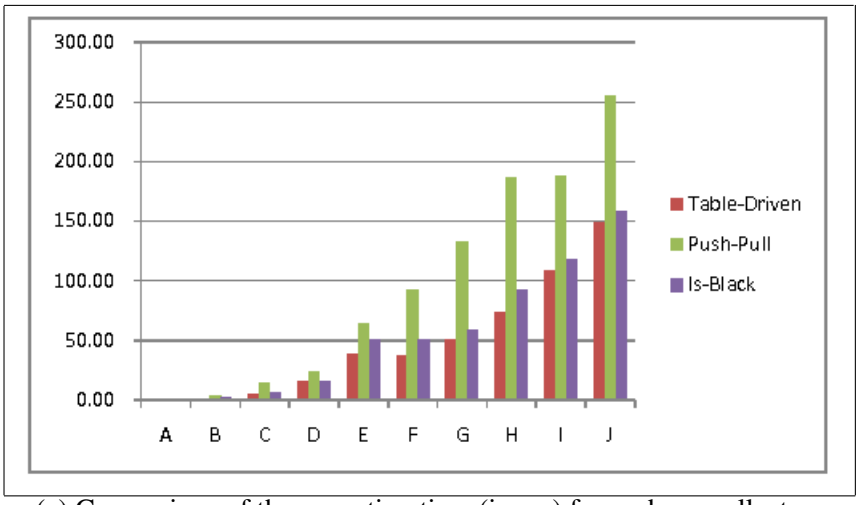
We experiment on the efficiency of the garbage collection algorithms with 10 test cases whose numbers of actors are ranged from 300 to 10000. The data of the test cases is listed in Figure **??**.

The result of the experiments on the test cases is listed in Figure **??**. The comparison of the execution time for the garbage collection algorithms is shown in Figure **??**(a), and the execution time reduction by the Table-Driven algorithm is shown in Figure **??**(b). As Figure **??** shows, the execution time of the garbage collection is reduced from 2.128% to 64.98% by the Table-Driven algorithm. More precisely,

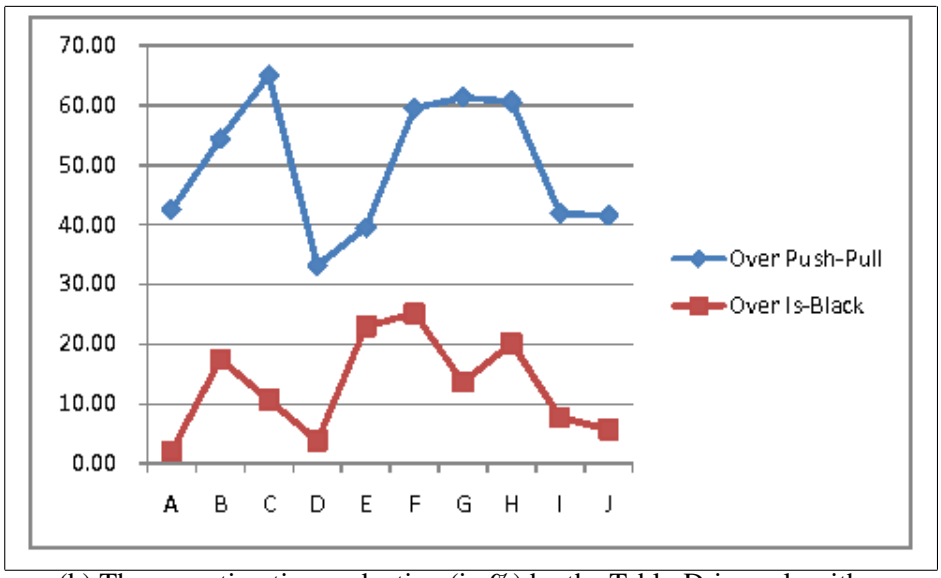
- It is reduced from 33.191% to 64.98% over the Push-Pull algorithm, averaged at 49.924%, and
- It is reduced from 2.128% to 25.129% over the Is-Black algorithm, averaged at 12.92%.

6. Conclusion

In this paper, we present a new garbage collection approach for active objects, called the Table-Driven algorithm. The algorithm reclaims the unused objects based on the garbage defined by Kafura et al. Improved from the Push-Pull algorithm and the Is-Black algorithm, our algorithm uses two auxiliary tables to



(a) Comparison of the execution time (in ms) for garbage collectors



(b) The execution time reduction (in %) by the Table-Driven algorithm

Figure 6: Comparison of the execution time for the garbage collection algorithms running on GCofActorsSimulator.

eliminate the redundant repeated visits to actors. We experiment on the efficiency of the garbage collection algorithms on an actor system simulator, called GCofActorsSimulator. According to the experiment result, the execution time of garbage collection is reduced from 2.128% to 64.98% by our new Table-Driven algorithm; more precisely, 33.191% to 64.98% over the Push-Pull algorithm, and 2.128% to 25.129% over the Is-Black algorithm.

References

[] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[] D. F. Bacon, P. Cheng, and D. Grove. "Garbage Collection for Embedded Systems," *Proceedings of the 4th ACM International Conference on Embedded Software* (EMSOFT '04), pp. 50-68, October 2004.

[] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. "Adaptive Garbage Collection for Battery-Operated Environments," *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pp1-12, 2002.

[] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. "Tuning Garbage Collection in an Embedded Java Environment," *Proceedings of Eighth International Symposium on High-Performance Computer Architecture*, pp. 92-103, February 2002.

[] P. Dickman. "Efficient, Incremental, Distributed Orphan Detection and Actor Garbage Collection using Graph Partitioning and Euler Cycles," *Proceedings of International Workshop on Distributed Algorithms and Graphs* (WDAG'96 now known as DISC), also published as LNCS 1151, pp. 141-158, October 1996.

[] R. R. Fenichel, and J. C. Yochelson. "A LISP Garbage Collector for Virtual-Memory Computer Systems," *Communication of ACM*, vol. 12, no. 11, pp. 611-612, November 1969.

[] B. Goldberg, and M. Gloger. "Polymorphic Type Reconstruction for Garbage Collection without Tags," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 53-65, June 1992.

[] R. Jones, and R. Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons Press, 1996.

[] D. Kafura, and K. Lee. "ACT++: Building A Concurrent C++ With Actors," Technical Report 89-18, Department of Computer Science, Virginia Tech., 1989.

[] D. Kafura, M. Mukherji, and D. M. Washabaugh. "Concurrent and Distributed Garbage Collection of Active Objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 337-350, April 1995.

[] D. Kafura, D. Washabaugh, and J. Nelson. "Garbage Collection of Actors," *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications* (OOPSLA'90), also appears at *ACM SIGPLAN Notices*, vol. 25, no.10, pp. 126-134, October 1990.

[] W. Y. Kim, and G. Agha. "Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages," *Proceedings of the IEEE/ACM Supercomputing Conference* (SC95), pp. 39, December 1995.

[] J. McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Communication of ACM*, vol. 3, no. 4, pp 184-195, 1960.

[] J. Nelson. *Automatic, Incremental, On-the-fly Garbage Collection of Actors*, M.S. Thesis, Department of Computer Sceince, Virginia Tech., February 1989.

[] Y. Park, and B. Goldberg. "Static Analysis for Optimizing Reference Counting," *Information Processing Letters*, vol. 55, no. 4, pp. 229-234, August 1995.

[] D. Plainfosse, and M. Shapiro. "A Survey of Distributed Garbage Collection Techniques," *Proceedings of International Workshop on Memory Management,* LNCS vol. 986, pp. 211-249, September 1995.

[] I. Puaut. "Distributed Garbage Collection of Active Objects with No Global Synchronisation," *Proceedings of International Workshop on Memory Management*, also published as LNCS 637, pp. 148-164, September 1992.

[] A. Vardhan. *Distributed Garbage Collection of Active Objects: A Transformation and Its Applications to Java Programming*, M.S. Thesis, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1998.

[] A. Vardhan, and G. Agha. "Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects," *Proceedings of the 3rd International Symposium on Memory Management*, pp. 106-113, June 2002.

[] W.-J. Wang, and C. A. Varela. “Distributed Garbage Collection for Mobile Actor Systems: the Pseudo Root Approach,” it Proceedings of International Conference on Grid and Pervasive Computing (GPC2006), also published as LNCS 3947, pp. 360-372, 2006.