A Fully Concurrent Garbage Collector for Functional Programs on Multicore Processors

Katsuhiro Ueno   Atsushi Ohori

Research Institute of Electrical Communication,
Tohoku University, Japan
{katsu, ohori}@riec.tohoku.ac.jp

Abstract

This paper presents a concurrent garbage collection method for functional programs running on a multicore processor. It is a concurrent extension of our bitmap-marking non-moving collector with Yuasa's *snapshot-at-the-beginning* strategy. Our collector is *unobtrusive* in the sense of the Doligez-Leroy-Gonthier collector; the collector does not stop any mutator thread nor does it force them to synchronize globally. The only critical sections between a mutator and the collector are the code to enqueue/dequeue a 32 kB allocation segment to/from a global segment list and the write barrier code to push an object pointer onto the collector's stack. Most of these data structures can be implemented in standard lock-free data structures. This achieves both efficient allocation and unobtrusive collection in a multicore system. The proposed method has been implemented in SML#, a full-scale Standard ML compiler supporting multiple native threads on multicore CPUs. Our benchmark tests show a drastically short pause time with reasonably low overhead compared to the sequential bitmap-marking collector.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Languages, Performance

Keywords Concurrent Garbage Collection, Multicore Processors, Standard ML, Functional Languages

1. Introduction

Today, multicore CPUs have become commodity hardware and multicore processors have become widespread in various computer platforms. This trend is likely to be more evident in the future – it is now a common belief that performance in future computer systems can be improved by increasing the number of cores in a single chip, rather than increasing the operating frequency. Along these lines, to develop more capable and efficient computer systems, it is essential to exploit multicore processors. Thread libraries in a number of operating systems, such as POSIX threads, already support multicore processors. We expect that rapidly progressing thread scheduling researches will make thread libraries efficient tools in diverse areas of application development for multicore and future many-core systems, including those of real-time embedded systems.

In order for functional programming to become a viable framework in future application development, it is essential to develop functional language supports for multicore processors. This includes both language constructs to express concurrent and parallel computation, and language support to use OS-supplied multicore-capable thread libraries. There are a number of proposals for the former, but relatively few have been studied for the latter. This paper is concerned with the latter problem. We note that there is a well taken argument that a functional language should provide light-weight threads efficiently implemented by the language run-time system. However, these language-level threads alone apparently cannot be scaled to multicore processors; they need to be implemented on top of OS-level native threads. Design and implementation of language-level light-weight threads are largely orthogonal to language support for native threads; therefore, as this issue is not the focus of this paper, we do not discuss it in this paper.

One major technical challenge in using native threads in functional programs is the development of a concurrent garbage collection method. Threads in a functional language require not only CPU time slices allocated by an OS thread scheduler but also heap memory allocated by a garbage collector. In order for functional programs to enjoy the complete benefits of native threads on a multicore processor, we have to develop a garbage collector that works with multiple native threads, each of which runs concurrently on a different core. Furthermore, we believe that if concurrent garbage collection is *unobtrusive* in the sense of [9, 10], i.e., if mutators are not required to pause due to memory allocation and collection, then it can open up the possibility of using a functional language in real-time system development. This is based on the following observation. A real-time system consists of a set of threads, some of which are deadline critical real-time ones. An OS thread scheduler realizes real-time scheduling by assigning priority to real-time threads in allocating time slices of the processor cores. Since heap memory is also a global system resource, a similar mechanism is required in heap memory allocation. An unobtrusive garbage collection method can provide real-time threads with the required heap memory by introducing some mechanism to give priority to real-time threads in allocating heap memory. Assuming that the garbage collection speed is fast enough to satisfy the memory requirement of real-time threads, the only overhead of an unobtrusive garbage collection method imposed on a priority-given real-time thread is the time required to handshake with the collector and to enumerate the thread-local root set. The former is negligible. The root set enumeration time is determined by the program code, and we expect that a real-time thread is coded in such a way that this time is within the acceptable bound at any execution point. Thus, the execution time of the high-priority thread can be predictable, satisfying the real-time requirement. This paper aims to develop such a garbage collection method and to demonstrate its feasibility by implementation and evaluation.

There have been some attempts to design and implement concurrent garbage collection algorithms. For a practical functional language implementation, however, very few unobtrusive algorithms have been developed and implemented. Blelloch and Cheng proposed a parallel and concurrent garbage collection [4] and implemented it in a Standard ML compiler [5], where the collector concurrently replicates a snapshot of the entire heap. One limitation of this method is that its collection is not unobtrusive; this method requires barrier synchronization of all the mutators to update the pointers in their thread-local variables (registers and active stacks). This implies that thread pause time is at least as long as the sum of barrier synchronization and the maximum time needed for some mutator to update its local variables. Although the authors developed a series of techniques to minimize the local-variable update overhead, considering this pause time to be a constant may be problematic, and in multicore and future many-core processors this overhead can be a serious bottleneck in an application consisting of a large number of threads. The only unobtrusive collector for functional language that we are aware of is the concurrent collector proposed and implemented by Doligez, Leroy, and Gonthier [9, 10], which we refer to as the Doligez-Leroy-Gonthier collector. This collector uses a variant of tricolor marking described by Dijkstra et al. [8]. The unobtrusiveness is realized by an ingenious phase delimiting handshaking between mutators and the collector. The authors showed some performance results on a prototype implementation of a bytecode interpreter for ML on a multi-processor machine; however, the reported performance data were rather limited and therefore its overhead compared with a conventional sequential garbage collection method in a full-scale native code compiler is unclear. For Java, Pizlo et al. developed a concurrent, unobtrusive, and real-time garbage collector [18]. Its unobtrusiveness is realized by a combination of fragmented allocations and concurrent replication of indirect pointers. Although the performance results are promising for the Java language, its effectiveness in allocation-intensive functional programs remains to be investigated. In addition, the indirect pointers and fragmented allocations are problematic in achieving interoperability with the C language, which is essential for using native thread libraries provided by the operating system.

A potentially serious problem in concurrent unobtrusive collection is allocation efficiency. The Doligez-Leroy-Gonthier collector does not fully address efficient allocation in a global heap. This and perhaps most other collectors so far implemented for functional languages on multicore systems use some form of thread-local heaps, or nurseries. The rationale underlying this strategy would be the consideration that functional programs require large amounts of short-lived data, for which generational copying collection (including mark-sweep collector with a nursery) would be the only option. It is indeed true that functional programs produce large amounts of short-lived objects very rapidly, and this also holds for multithread programs. However, for a concurrent program manipulating a large shared data structure to scale to a number of threads, efficient allocation in a global heap is also essential. If each mutator thread initially allocates objects in a thread-local heap, then frequent and costly promotion of thread-local data to global heap is required. There have been some attempts, notably [15], to minimize this local-global promotion overhead, but a certain amount of non-trivial overhead seems to be inevitable. This problem can be avoided if we have a collection method that efficiently allocates objects directly in a non-moving global heap and then we extend it to fully concurrent unobtrusive collection without introducing much additional overhead; this paper focuses on the latter development.

We base our development on our non-moving collector for functional languages [20], whose overall performance is roughly comparable to a generational copying collector. In the sequel, we refer to it as the Ueno-Ohori-Otomo collector. This collector has the following general structure:

1. Objects are allocated in a single global heap managed by a set of segments (BiBoP).
2. Each segment contains its own allocation pointer, a bitmap tree representing object liveness, and a working area for the trace stack. For generational collection, the remembered set is also allocated in the working area.
3. It avoids compaction entirely by organizing the heap as a collections of sub-heaps $\{H_i \mid c \leq i \leq B\}$ of exponentially increasing allocation sizes (H_i for 2^i bytes).

We observe that this structure provides an ideal basis for multithread extension, and that integrating it with Yuasa's snapshot-at-the-beginning [21] algorithm yields a desired unobtrusive collection method suitable for multithread functional programs on multicore processors.

We realize a concurrent and unobtrusive collection roughly by the following strategy. The multithread extension of the Ueno-Ohori-Otomo collector is straightforwardly obtained by assigning each thread its own allocation segments. This allows independent and simultaneous allocation in a single global heap. Allocation in the resulting concurrent collection method can be as efficient as in the sequential collection. We combine this multithread extension with the snapshot-at-the-beginning algorithm to realize a concurrent collection. When collection is triggered, the collector obtains a set of filled segments and handshakes with mutators to obtain their roots and starts write barriers. Then, it performs collection only on the filled segments. Using the handshaking method presented in [9], the roots of all mutators can be obtained without stopping the mutators. By extending the snapshot algorithm with a notion analogous to *sliding view* by Levanoni and Petrank [13], we can establish a provably correct concurrent snapshot abstraction. Moreover, since the memory demand of mutators can be measured by the number of filled and free segments, the collector automatically adjusts the number of segments and the size of sub-heaps to prevent memory starvation, which may pause real-time mutator threads.

We work out the various subtle details of this strategy and develop a new concurrent and unobtrusive collection method. Contributions made in this paper include the following:

- We developed a concurrent extension of Yuasa's snapshot abstraction and established its correctness. To do this, we adopted the snooping write barrier [13] and developed a concurrent snapshot algorithm for multiple mutators. We presented a direct and simple correctness proof for the algorithm.
- We fully implemented the collection method in a full-scale Standard ML compiler. We choose SML# [19] because of its native thread support on multicore processors.
- We measured the overhead due to concurrent collection by comparing the performance of our concurrent collector on single-thread ML programs to that of a well-tuned sequential collector. We obtained the following results. Overhead of our concurrent collector is about 12% compared with the sequential one. Compared with existing work on concurrent collection for functional languages, this figure is quite satisfactory.
- Our concurrent garbage collection significantly reduces maximum pause time.

The rest of the paper is organized as follows. Section 2 outlines the two basic collection methods on which this work is based.

Section 3 presents our collection method. Section 4 reports implementation details and addresses some practical issues that need to be overcome during the implementation. Section 5 reports performance evaluation. Section 6 compares the contribution with related works. Section 7 concludes the paper.

2. Basic Garbage Collection Algorithms

In this section, we first outline the structure and algorithm of the Ueno-Ohori-Otomo collector to the extent relevant to our concurrent extension, and then present Yuasa's snapshot-at-the-beginning algorithm with some properties that are useful in developing a new concurrent abstraction.

2.1 Ueno-Ohori-Otomo Collector

In this collector, a given allocation space is divided into a set of allocation segments. A segment contains an array of allocation blocks of the same size. The segment size is a tuning parameter. For concurrent extension, we choose 32 kB for the segment size.

The heap is set up as a family of sub-heaps of various sizes:

$$(\mathit{Free}, (H_3, \ldots, H_{12}, \ldots))$$

where *Free* is the free segment list, i.e., the list of unused segments, and each H_i is a sub-heap for 2^i-byte allocation blocks. Setting up the heap as a family of sub-heaps of exponentially increasing allocation sizes is essential in reducing internal fragmentation. For the purpose of presenting the concurrent extension, however, it is enough to consider a set of sub-heaps $H_3, \ldots, H_{12}, \ldots$ as a single allocation heap H for a single block size.

A set of segments belonging to H is organized as

$$(\mathit{Filled}, \mathit{Current}, \mathit{Active})$$

where *Filled* is the list of segments already filled, *Current* is the segment into which objects are currently allocated, and *Active* is the list of segments partially filled.

Each segment has the following structure:

$$(\mathit{Blks}, \mathit{AllocPtr}, \mathit{Bitmap})$$

where *Blks* is an array of allocation blocks of the same size, *AllocPtr* is a pointer (an abstract data structure) to the next block to be allocated, and *Bitmap* is a (tree structured) bitmap that represents object liveness. Each (leaf) bit in *Bitmap* represents the mark bit of corresponding allocation block in *Blks*. Objects are allocated in blocks in *Blks* linearly from left to right by advancing *AllocPtr* to the position where the mark bit is not set. The ingenious organization of *AllocPtr* and *Bitmap* makes this free block search efficient. Note that allocation does not set the corresponding mark bit.

For the aforementioned settings, allocating an object in H is achieved as follows:

1. Attempt to find a free block in *Current* and, if found, return it.
2. If *Current* is full, i.e., *AllocPtr* reaches the end of *Blks*, place *Current* to *Filled* and attempt to obtain a segment from *Active*. If this succeeds, set it as *Current* and try step 1 again.
3. If *Active* is empty, attempt to obtain a segment from *Free*. If this succeeds, set it as *Current* and try step 1 again.
4. If *Free* is empty, invoke the garbage collector and try step 2 again.

The garbage collector performs the following operations:

1. It clears all the bitmaps and rewinds all the allocation pointers of all the segments.
2. It obtains the root set from the mutator.
3. It traces the set of live objects reachable from the root set. During the trace, it sets the mark bit of each live object in the corresponding bitmap tree.
4. At this point, all the bitmaps reflect the exact liveness. The collector then scans the segments and places each of them in one of the lists: *Filled*, *Active*, or *Free*, depending on the number of live objects in the segment.

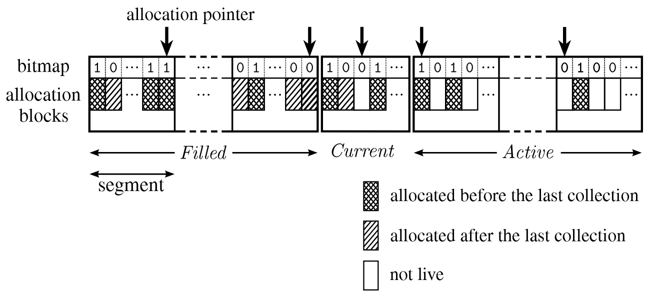

Figure 1. Structure of an allocation heap

From this structure and the linear allocation strategy, the following invariants hold:

- Marked objects were allocated before the last collection.
- The unmarked objects in *Filled* and those in the blocks up to *AllocPtr* in *Current* were allocated after the last collection.
- The unmarked objects in *Active* and those in the blocks after the *AllocPtr* in *Current* are not live.
- The allocation pointer of a segment in *Filled* points to the end of the block array, and that in *Active* points to the beginning.

Figure 1 shows the structure of an allocation heap with the time when each object is allocated.

From these invariants, the following property holds:

> Let t be any given time. At any time after t, the collector can determine the set of allocated objects at t if the collector has a snapshot of all the allocation pointers at t.

This property makes the Ueno-Ohori-Otomo collector a suitable basis for realizing Yuasa's snapshot-at-the-beginning method, since taking a snapshot can be done just by saving allocation pointers with very low cost.

2.2 Yuasa's Snapshot Abstraction

Yuasa's snapshot method [21] was originally presented for real-time garbage collection where a collector and mutator interleave, but it can also be used as a basis for concurrent (on-the-fly) garbage collection. In the literature, this abstraction is sometimes explained using tricolor marking abstraction. However, it is more direct and simpler to explain this method if we view Yuasa's idea of taking a snapshot as the central abstraction. The snapshot abstraction simply marks and collects the "snapshot" of the heap, i.e., the set of live objects *at the instant when collection began.* For the purpose of our development, we formally state this abstraction below.

We model a heap as a set of objects (taken from a fixed countably infinite universe of objects ranged over by o) equipped with a binary relation of reachability. For any time t, we write $\mathbf{H}(t)$ for the set of objects in the heap at t. We write $o_1 \rightarrow_t o_2$ if o_1 contains a pointer to o_2 in $\mathbf{H}(t)$. In the following abstract arguments, we regard $\mathbf{H}(t)$ to be monotonically increasing with respect to time t. We write $\mathcal{R}(\mathbf{H}(t))(S)$ for the set of objects reachable in $\mathbf{H}(t)$ from some member of S. In this notation, we allow S to contain some objects not in $\mathbf{H}(t)$.

Let t_0 be the time when collection begins. Let $R(t_0)$ be the root set at t_0. The snapshot abstraction reclaims the following set of objects

$$\mathbf{G} = \mathbf{H}(t_0) \setminus \mathcal{R}(\mathbf{H}(t_0))(R(t_0))$$

at some time later than t_0. This is all that this abstraction states. Since any future reference into $\mathbf{H}(t_0)$ is in $\mathcal{R}(\mathbf{H}(t_0))(R(t_0))$, this is obviously correct by this construction, regardless of any future (concurrent) modification to the heap by a single or multiple mutators.

In our development, it is beneficial to make this obvious statement explicit as follows. Let t be some time later than t_0, written as $t \geq t_0$. The soundness of snapshot abstraction is the following property:

$$\mathbf{G} \cap \mathcal{R}(\mathbf{H}(t))(R(t)) = \emptyset$$

This is guaranteed by

$$\mathcal{R}(\mathbf{H}(t))(R(t)) \cap \mathbf{H}(t_0) \subseteq \mathcal{R}(\mathbf{H}(t_0))(R(t_0))$$

In order to implement the snapshot abstraction, we only need to

- preserve the reachability relation $\mathcal{R}(\mathbf{H}(t_0))$, and
- determine whether or not an object is in $\mathbf{H}(t_0)$ at any time $t \geq t_0$.

The former is achieved by remembering all the pointers that are overwritten by a mutator later than t_0. The latter is achieved through the organization of the heap.

3. Concurrent Unobtrusive Collection

We extend the Ueno-Ohori-Otomo collector to a concurrent unobtrusive collection that works on multicore systems by combining it with Yuasa's snapshot abstraction. The development is presented in the following steps. First, we introduce two disjoint extensions of the Ueno-Ohori-Otomo collector: the snapshot extension for a single mutator and concurrent allocation for multithread support. Second, we develop the concurrent snapshot abstraction for multiple mutators and demonstrate its correctness. As a refinement to the abstraction, we address the following three subtle issues: (1) determining the snapshot time, (2) object initialization, and (3) root set enumeration. Finally, we present the overall algorithm of the concurrent unobtrusive collection method by combining all the above developments.

3.1 Snapshot Extension For a Single Mutator

Consider a case where there are a single mutator and a collector. We assume that the collector is invoked by some trigger. The collector first requests the mutator to send its root set. When the single mutator receives the request, it stops its execution at time t_0. We choose t_0 as the time when the snapshot is taken. The mutator then saves its registers and scans its stack frames to obtain its root set at t_0, written as $R(t_0)$. To implement the snapshot abstraction, we additionally need to determine the set $\mathbf{H}(t_0)$ and the relation $\mathcal{R}(\mathbf{H}(t_0))$ at any future time $t \geq t_0$.

In the Ueno-Ohori-Otomo collector, $\mathbf{H}(t_0)$ is the set of all objects in *Filled*, marked objects in *Current* and *Active*, and unmarked objects in *Current* before *AllocPtr* at time t_0. As we mentioned in Section 2.1, this can be saved by taking a snapshot of allocation pointers. To record the pointer snapshots, we extend the segment structure by adding a saved pointer field as follows:

$$(\mathit{Blks}, \mathit{AllocPtr}, \mathit{AllocPtrSnap}, \mathit{Bitmap})$$

where *AllocPtrSnap* is the area to hold a snapshot of *AllocPtr*. *AllocPtrSnap* is initialized with the pointer to the beginning of *Blks*. At time t_0, for every segment in *Filled* and *Current*, *AllocPtr* is copied to *AllocPtrSnap*. Consequently, at any future time $t \geq t_0$, the predicate $o \in \mathbf{H}(t_0)$ holds if either o is marked or the address of o is less than *AllocPtrSnap* of the segment to which o belongs.

For the collector to determine the set $\mathcal{R}(\mathbf{H}(t_0))(R(t_0))$, from the time t_0 until the time the collector finishes tracing, the mutator executes snapshot write barrier `remember(!`x`)` just before every pointer mutation $x := y$[1]. The barrier `remember(`p`)` inserts pointer p into the global remembered set $\mathbf{R}$. Consequently, $\mathbf{R}$ saves original object reachability at t_0 that are overwritten by mutations after t_0. The collector determines (a superset of) the set $\mathcal{R}(\mathbf{H}(t_0))(R(t_0))$ by tracing objects from $R(t_0)$ and $\mathbf{R}$ but only through the objects in $\mathbf{H}(t_0)$. After $\mathcal{R}(\mathbf{H}(t_0))(R(t_0))$ is determined, we can safely reclaim any unreachable object in $\mathbf{H}(t_0)$. For simplicity, we reclaim those in *Filled* at time t_0 instead of those in $\mathbf{H}(t_0)$.

The snapshot collection algorithm is summarized as follows:

1. The collector fixes the set of segments to be reclaimed at this collection cycle by moving segments in *Filled* to a set $\mathbf{C}$, which is a set of segments managed only by the collector.
2. The collector requests the mutator to send the root set. The mutator then pushes its root set to the collector's trace stack, saves allocation pointer to *AllocPtrSnap* for each segment in *Filled* and *Current*, and turns on the snapshot write barrier.
3. For each segment in $\mathbf{C}$, the collector clears its bitmap and saves its allocation pointer to *AllocPtrSnap*. The collector performs this without any exclusive lock since segments in $\mathbf{C}$ are managed only by the collector.
4. After the mutator turns on the snapshot write barrier, the collector traces and marks objects reachable from the root set and the remembered set $\mathbf{R}$ by repeating the following procedure:
   (a) Obtain an object o from the trace stack or the remembered set $\mathbf{R}$.
   (b) If both are empty, the collector finishes its tracing and marking.
   (c) If o is already traced then go to the next iteration.
   (d) If either o is marked or the address of o is less than *AllocPtrSnap*, mark o and push all pointers in o to the trace stack.
5. At any time after the tracing is finished, the mutator can turn off the snapshot write barrier.
6. The collector moves each segment in $\mathbf{C}$ to either *Filled*, *Active*, or *Free*, depending on the number of live objects in it. When a segment is placed in *Active*, its *AllocPtr* and *AllocPtrSnap* are initialized.

Note that the mutator keeps adding pointers to $\mathbf{R}$ while the collector takes pointers out from $\mathbf{R}$ during the tracing. This is the only source of contention in this algorithm, which is theoretically inherent. As mentioned later, we implement $\mathbf{R}$ as a separate linked list to reduce the contention.

3.2 Concurrent Allocation

In the Ueno-Ohori-Otomo collector, an object is always allocated in the current segment *Current*. To support multiple concurrent mutator threads $\{M^j \mid 1 \leq j \leq n\}$, we assign a separate current

[1] We use the ML-like notation. Here, x denotes a pointer to a pointer field. $x := y$ updates the pointer field pointed by x with pointer y. `!`x is the pointer stored in the pointer field pointed by x.

segment $Current^j$ to each thread M^j. The structure of heap H now becomes the following:

$$(Free, (Filled, \{Current^1, \ldots, Current^n\}, Active))$$

Since each segment has its own allocation information including the allocation pointer, each M^j can concurrently allocate objects in its current segment $Current^j$ without any mutual exclusion. *Free*, *Filled*, and *Active* are shared by the mutators and collector. We implement them as lock-free stacks protected by compare-and-swap atomic operation. The allocation operation is the same as those in Section 2.1 except that, in step 2 and 3, the allocator in each thread accesses one of these lists through atomic operations.

For this heap structure, the stop-the-world collector can be easily obtained by inserting the barrier synchronization of all the running mutator threads before starting the collection. As reported in Section 5, we implemented this collector for comparison purposes. While it has the overhead of stopping all the threads, this collector is very fast, especially in allocation, and as it stands, it is one candidate collector for multithread functional programs. However, when the number of mutator threads increases to, say 100 or 1000, whose numbers are not unrealistic, thread synchronization overhead imposed on each collection cycle would become a major obstacle for concurrent/parallel functional programs. For this purpose, we would like to develop an efficient unobtrusive concurrent collector for multiple mutators, to which we now turn.

3.3 Concurrent Snapshot Abstraction for Multiple Mutators

As observed by Levanoni and Petrank [13], when there is more than one mutator, it is not possible to determine the time t_0 when the snapshot is made. Since $\mathbf{H}(t_0)$ and $\mathcal{R}(\mathbf{H}(t_0))$ depend on this time t_0, the snapshot abstraction is not directly applicable for multiple mutators. To deal with this problem, Levanoni and Petrank proposed a new abstraction called *sliding view* to be used with snapshot abstraction. This abstraction carefully eliminates the anomalous behavior that arises with multiple concurrent mutators. We largely follow this proposal. However, we provide a new abstraction that is directly based on the Yuasa's original observation of snapshot abstraction. This new abstraction yields a remarkably simple correctness proof for concurrent snapshot garbage collection. Since the required write barrier operation is essentially the same, the abstraction presented here would also shed some light on the sliding view abstraction or other concurrent extensions of snapshot algorithms.

We consider a case where there are two mutators M_1 and M_2. Generalization to multiple mutators is straightforward. Let t_1 and t_2 be two points in time such that $t_1 \leq t_2$. Suppose that at time t_1, M_1 obtains its root set $R_1(t_1)$, and at time t_2, M_2 obtains its root set $R_2(t_2)$. Let $R_1(t_2)$ be the root set of M_1 at t_2 and $R_2(t_1)$ be that of M_2 at t_1. The reclamation will be performed at some time after t_2.

The snapshot abstraction allows us to reclaim the following set of objects at any time after t_1:

$$\mathbf{G}_1 \quad = \quad \mathbf{H}(t_1) \setminus \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1) \cup R_2(t_1))$$

As we shall show below, using a handshaking mechanism similar to the one proposed in [9], it is possible to obtain $\mathbf{H}(t_1)$ by turning on the snapshot write barriers before t_1. Hence, if the set $R_1(t_1) \cup R_2(t_1)$ would be available, we see that we can collect $\mathbf{G}$. Unfortunately, however, the available set is only $R_1(t_1) \cup R_2(t_2)$.

To fill the gap, we exploit the following additional general property on the snapshot abstraction: from the soundness argument of the snapshot abstraction, instead of$\mathbf{G}_1$, the following set can safely be reclaimed at any time after t_2:

$$\mathbf{G}_2 \quad = \quad \mathbf{H}(t_1) \setminus \mathcal{R}(\mathbf{H}(t_2))(R_1(t_2) \cup R_2(t_2))$$

This is equivalent to the following:

$$\begin{aligned} \mathbf{G}_2 &= \mathbf{H}(t_1) \setminus \mathbf{T}_2 \\ \mathbf{T}_2 &= \mathcal{R}(\mathbf{H}(t_2))(R_1(t_2) \cup R_2(t_2)) \cap \mathbf{H}(t_1) \end{aligned}$$

Let $\mathbf{I}(t_1, t_2)$ be the set of instructions with their actual operands executed from t_1 to t_2 by both M_1 and M_2. For simplicity, here, we consider object allocation as a sequence of allocation with null pointers followed by the mutations with the initial values. We then have the following:

LEMMA 1.

$$\begin{aligned} & \mathcal{R}(\mathbf{H}(t_2))(R_1(t_2) \cup R_2(t_2)) \cap \mathbf{H}(t_1) \\ \subseteq\ & \mathcal{R}(\mathbf{H}(t_1))(R_1(t_2) \cup R_2(t_2) \cup \mathbf{S}) \\ & \textit{where } \mathbf{S} = \{y \mid x := y \in \mathbf{I}(t_1, t_2), y \in \mathbf{H}(t_1)\} \end{aligned}$$

Proof. Let o be any element in $\mathcal{R}(\mathbf{H}(t_2))(R_1(t_2) \cup R_2(t_2)) \cap \mathbf{H}(t_1)$. By definition, there is a sequence of point-to relation chain $o_1 \rightarrow_{t_2} \ldots \rightarrow_{t_2} o_n$ in $\mathbf{H}(t_2)$ such that $o_1 \in R_1(t_2) \cup R_2(t_2)$, $o_n \in \mathbf{H}(t_1)$, and $o = o_n$. If $o_1 \rightarrow_{t_1} \ldots \rightarrow_{t_1} o_n$ also holds, $o_n \in \mathcal{R}(\mathbf{H}(t_1))(R_1(t_2) \cup R_2(t_2))$. Else, there is some j such that $o_1 \rightarrow_{t_2} \cdots \rightarrow_{t_2} o_j \not\rightarrow_{t_1} o_{j+1} \rightarrow_{t_1} \cdots \rightarrow_{t_1} o_n$. Since $o_j \notin \mathbf{H}(t_1)$ but $o_{j+1} \in \mathbf{H}(t_1)$, this can only happen by the mutation of some of the fields of o_j with o_{j+1}. The desired result then follows. □

By this lemma, we can trace the following set $\mathbf{T}_3$ instead of $\mathbf{T}_2$:

$$\begin{aligned} \mathbf{T}_3 &= \mathcal{R}(\mathbf{H}(t_1))(R_1(t_2) \cup R_2(t_2) \cup \mathbf{S}) \\ \mathbf{S} &= \{y \mid x := y \in \mathbf{I}(t_1, t_2), y \in \mathbf{H}(t_1)\} \end{aligned}$$

Furthermore, for the set $\mathbf{S}$ defined above, we have the following.

LEMMA 2.

$$\mathcal{R}(\mathbf{H}(t_1))(R_1(t_2)) \subseteq \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1) \cup \mathbf{S})$$

Proof. We first observe the following: since $\mathcal{R}(\mathbf{H}(t_1))(R_1(t_1))$ is the set of all reachable objects at time t_1 by the mutator M_1, if there is some pointer o in $\mathcal{R}(\mathbf{H}(t_1))(R_1(t_2))$ but not in $\mathcal{R}(\mathbf{H}(t_1))(R_1(t_1))$, there must be some $o' \in \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1))$ such that o is not reachable from o' in $\mathbf{H}(t_1)$ but reachable in $\mathbf{H}(t)$ for some t such that $t_1 < t \leq t_2$. This is possible only by some mutation with o' in its operand performed by another mutator. The result then follows by a similar argument as in Lemma 1. □

This result shows that the following set can be traced instead of $\mathbf{T}_3$:

$$\mathbf{T}_4 \quad = \quad \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1) \cup R_2(t_2) \cup \mathbf{S})$$

The snapshot abstraction for two mutators is now stated as follows:

THEOREM 1 (Snapshot abstraction with two mutators). *Let* t_1, t_2 *be two points in time and* $R_1(t_1)$ *and* $R_2(t_2)$ *be the root sets of two mutators at* t_1 *and* t_2*, respectively. Let*

$$\begin{aligned} \mathbf{G} &= \mathbf{H}(t_1) \setminus \mathbf{T} \\ \mathbf{T} &= \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1) \cup R_2(t_2) \cup \mathbf{S}) \\ \mathbf{S} &= \{y \mid x := y \in \mathbf{I}(t_1, t_2), y \in \mathbf{H}(t_1)\}, \end{aligned}$$

then for any time t after t_2*,*

$$\mathbf{G} \cap \mathcal{R}(\mathbf{H}(t))(R_1(t) \cup R_2(t)) = \emptyset$$

Proof. From Lemma 1 and Lemma 2. □

This implies that the collector can request and receive the root sets $R_1(t_1)$ of mutator M_1 and $R_2(t_2)$ of mutator M_2 at different time points t_1 and t_2, provided that the snoop set $\mathbf{S}$ is recorded during the time interval $[t_1, t_2]$. This interval corresponds to the snooping period of sliding view in [13]. We note that our proof above is a direct and purely semantic one that holds for any collection algorithms that conform to this abstraction.

By generalizing the above argument to multiple mutators, we obtain the following fully concurrent snapshot abstraction. Let $M_1, \ldots, M_n$ be a set of mutators, and $t_1, \ldots, t_n$ be the time points at which each mutator M_i takes its root set. We assume $t_1 \leq \cdots \leq t_n$ without loss of generality. Since each mutator does not know the order of the time points $t_1, \ldots, t_n$, we need to define the snoop set as

$$\mathbf{S} \quad = \quad \{y \mid x := y \in \mathbf{I}(t_1, t_n)\}$$

where we record all the pointers that are assigned during the entire time interval $[t_1, t_n]$ which can be delimited by handshaking. The fully concurrent snapshot abstraction reclaims the following set $\mathbf{G}$:

$$\begin{array}{rcl} \mathbf{G} & = & \mathbf{H}(t_1) \setminus \mathbf{T} \\ \mathbf{T} & = & \mathcal{R}(\mathbf{H}(t_1))(R_1(t_1) \cup \ldots \cup R_n(t_n) \cup \mathbf{S}) \end{array}$$

where $\mathbf{S}$ is the set defined above. For this, we have the following:

COROLLARY 1. $\mathbf{G}$ *is not reachable from any root set $R_i(t)$ at any time t after t_n.*

3.4 Determining t_0, $\mathbf{H}(t_0)$, and Root Sets

In order to implement the above unobtrusive snapshot abstraction, we need to determine the following things:

- The time t_0 at which a collection cycle starts.
- The set of objects in $\mathbf{H}(t_0)$.
- The reachability relation $\mathcal{R}(\mathbf{H}(t_0))$.
- The set $\{R_i(t_i) \mid 1 \leq i \leq n\}$ of root sets of all the mutators $\{M_i \mid 1 \leq i \leq n\}$, taken at some time points $t_1, \ldots, t_n$ later than t_0.
- The snoop set $\mathbf{S}$ defined above that must be taken over the time interval that covers all the time points $\{t_1, \ldots, t_n\}$.

t_0 can be any time earlier than any of the time points $t_1, \ldots, t_n$. Therefore, the collector determines t_0 at the beginning of its collection cycle. $\mathcal{R}(\mathbf{H}(t_0))$ is saved in the remembered set $\mathbf{R}$ by the snapshot write barrier. $\mathbf{S}$ is obtained by the snooping write barrier. To obtain $\mathbf{R}$ and $\mathbf{S}$, we turn on both the write barriers in all the mutator threads before time t_0.

What remains is $\mathbf{H}(t_0)$. In the snapshot abstraction for single mutator, this is determined by taking snapshots of allocation pointers; however, for multiple mutators, it is not possible to take those of all the mutators at the same time t_0. Because of the time lag in taking the snapshots in each mutator, the set of objects determined by the allocation pointer snapshots, written as S, may include objects allocated during the interval $[t_1, t_n]$ in addition to $\mathbf{H}(t_0)$. While S is not equivalent to $\mathbf{H}(t_0)$, we can safely use S instead of $\mathbf{H}(t_0)$ in the collection algorithm for the following reason. In the collection algorithm presented in Section 3.1, $\mathbf{H}(t_0)$ is used to determine two sets of objects: objects to be traced and objects to be reclaimed. Therefore, if we use a conservative set of $\mathbf{H}(t_0)$ for each purpose instead of $\mathbf{H}(t_0)$, the algorithm works correctly without an exact $\mathbf{H}(t_0)$. For tracing, S is a good alternative to $\mathbf{H}(t_0)$ since it covers $\mathbf{H}(t_0)$. For reclamation, we fix the collect set $\mathbf{C}$ at some time before t_0.

To force mutators to determine all the above things in every collection cycle, we design a handshaking mechanism similar to the one proposed in [9]. This also avoids the anomalous behavior of a barriered mutation beyond collection cycles shown in [9]. We introduce the following set of four global phases:

- SYNC1: this phase starts at the beginning of a collection cycle. In this phase, the mutators turn on their snooping and snapshot write barriers.
- SYNC2: this phase starts when the collector determines that all mutators turn on their write barriers. We fix t_0 at the moment of entering this phase. At the same time, the collect set $\mathbf{C}$ is taken as the set of segments in Filled. In this phase, the mutators send their root sets to the collector and saves allocation pointers in their $\mathit{Current}$.
- MARK: this phase starts when the collector receives root sets from all the mutators. During this phase, the collector traces objects reachable from the root sets and the remembered set maintained by the write barriers. In this phase, the mutators turn off their snooping write barriers at some time.
- ASYNC: this is the phase after finishing the tracing and before starting the next collection cycle. In this phase, the collector reclaims untraced objects. The mutators turn off their snapshot and snooping write barriers. This phase continues even after the reclamation completes until the next collection cycle starts.

The following global data structures are used to realize the handshaking:

- PHASE: A global flag indicating the global phase. This is updated by the collector and asynchronously read by each mutator.
- HANDSHAKE: A global counter that atomically counts up to the number of threads. The collector uses this to determine that all the mutators have detected the phase change.

In addition, each mutator has a local phase flag that reflects the phase each mutator is in. This is used to recognize the phase change and turn on write barriers.

The handshaking is performed as follows:

- A mutator periodically compares its local phase to PHASE. If different, it copies PHASE to its local phase. If the new phase is SYNC1, it increments HANDSHAKE. If the new phase is SYNC2, it pushes objects in its root set into the collector's stack and then increments HANDSHAKE.
- The collector iterates the following steps:
  1. When a collection cycle is triggered, it changes PHASE from ASYNC to SYNC1 and waits for HANDSHAKE to become full.
  2. When HANDSHAKE becomes full, it resets HANDSHAKE. Then it changes PHASE to SYNC2 and waits for HANDSHAKE to become full again.
  3. When HANDSHAKE becomes full, it resets HANDSHAKE. It then changes PHASE to MARK and start tracing. The tracing algorithm is similar to that in Section 2.1.
  4. After the tracing completes, it changes PHASE to ASYNC. The reclamation of segments in $\mathbf{C}$ is also same as that in Section 2.1.

3.5 Omitting Write Barriers at Object Initialization

In the abstract model in Section 3.3, we consider that objects are initialized by a sequence of mutations with snooping write barriers. This requires the compiler to insert write barrier codes to all object allocation points since the timing of garbage collection is not known statically. In functional programs, which often allocate tons of objects very frequently, the overhead of this write barrier code can be significantly large. For the best performance, we need to eliminate such write barriers from object initialization as much as possible without losing the safety of the snapshot collection.

We conclude that snooping write barriers at object initialization can be omitted for the following analysis of the usage of the snoop set. Let the left- and right-hand side of the statement of Lemma 1 be

$\mathbf{T}_2$ and $\mathbf{T}'_2$, respectively. $\mathbf{T}_2$ is the set of all live objects in $\mathbf{H}(t_1)$ at t_2. $\mathbf{T}'_2$ is a subset of $\mathbf{H}(t_1)$. Therefore, objects in $\mathbf{T}'_2 \setminus \mathbf{T}_2$ are garbage at any time after t_2. This means that $\mathbf{S}$ of Lemma 1 can safely be ignored in an actual system. In Lemma 2, the purpose of $\mathbf{S}$ is to capture mutations $x := y$ such that x is reachable from the root set of other threads. Hence, pointers added to S by $x := y$ can be safely ignored if x is not reachable from any other thread. A newly allocated object usually satisfies this condition since the object allocation code stores the pointer to a new object only in a thread-local variable. Consequently, we can safely omit write barriers at object initialization if every newly allocated object is initialized before the pointer to the new object is stored somewhere other than the thread-local variable.

3.6 Root Set Enumeration of Suspended Threads

In the abstract model, we simply assume that the set of threads is fixed and each thread keeps running. In an actual system, however, the set of threads may vary and a thread execution may be blocked to wait for some event such as I/O. For the former, we simply prohibit the thread creation and termination during SYNC1 and SYNC2. The latter induces a subtle issue; such waits may stop the handshaking of the collector and therefore all the mutators may be stuck owing to heap exhaustion. To avoid this, the collector must be able to determine the set of threads that would not respond to any handshake request and it needs to steal the root set of such threads.

To deal with these situations, we consider the frame stack and local phase flag of each thread as shared data structures protected by a binary semaphore. When a mutator thread is running under the garbage collection (GC) context, which is the usual case, the mutator thread holds this semaphore. When a thread leaves the GC context, it releases this semaphore. At the global phase transition to either SYNC1 or SYNC2, the collector tries to acquire the semaphore for each mutator thread. If it succeeds, the collector compares the local phase to PHASE and performs the required action on behalf of the blocked mutator. If failed, the collector does nothing for that thread and waits on HANDSHAKE counter. In this case, the mutator thread holding the semaphore performs the action.

The problematic case is that a mutator intends to leave the GC context at the same time the global phase is changed. Such a mutator must respond to the phase change before releasing the semaphore. To ensure the progress of the handshaking, all of the following operations by such a mutator must be atomically performed:

1. read the current PHASE,
2. copy it to the local phase flag, and
3. release the semaphore.

Otherwise, the mutator and collector may deadlock because the collector may change PHASE and wait on HANDSHAKE between these steps and thus the mutator would miss the phase change.

To avoid this deadlock, we reformulate the handshaking mechanism presented in Section 3.4 as a state machine that embodies the phase change and binary semaphore of a mutator. A state is a triple g-$\langle l$-$a\rangle$ indicating a combination of the current global phase g, a local phase flag l, and a status of binary semaphore a that is either LOCK or UNLOCK. For example, we write SYNC2-$\langle$SYNC1-LOCK$\rangle$ to indicate that the current global phase is SYNC2, the local phase flag is SYNC1, and the semaphore is acquired. There are only six meaningful combinations of PHASE and the local phase: ASYNC-$\langle$ASYNC$\rangle$, SYNC1-$\langle$ASYNC$\rangle$, SYNC1-$\langle$SYNC1$\rangle$, SYNC2-$\langle$SYNC1$\rangle$, SYNC2-$\langle$SYNC2$\rangle$, and MARK-$\langle$MARK$\rangle$; therefore, there are 12 possible states in total. The handshaking algorithm is elaborated as follows. Each thread has its own state instead of its local phase flag and semaphore. All parts of a state must be updated atomically; this can be realized efficiently by an atomic arithmetic or compare-and-swap operation. Before a handshake begins, every mutator thread is in ASYNC-$\langle$ASYNC-$a\rangle$ for some a. During handshaking, the collector changes the global phase parts of the states of all mutator threads instead of setting the PHASE flag. If a mutator detects that the local phase part of its state is not equal to its global phase part, it copies the global part to the local part and then performs an appropriate action. To leave the GC context, a mutator changes its state from g-$\langle l$-LOCK$\rangle$ to g-$\langle g$-UNLOCK$\rangle$. The detailed state transition, which is derived from the handshaking algorithm with the consideration of thread pause, is shown in the next subsection. We verify the correctness and progress of the elaborated algorithm using SPIN model checker.

3.7 Overall Algorithm

Following the discussion above, we now present the proposed fully concurrent unobtrusive garbage collection algorithm.

We use the following global data structures:

- HANDSHAKE, which is the same as in Section 3.4.
- NTHREADS: a counter holding the number of threads currently working.
- INITSTATE: a flag indicating the initial state of new threads.
- THREADLOCK: a binary semaphore protecting NTHREADS and INITSTATE.

In addition, each mutator thread has its own state flag that is updated atomically by the mutator and collector. INITSTATE is initialized to ASYNC-$\langle$ASYNC-LOCK$\rangle$.

The mutators' functions are as follows:

- Alloc: Object allocation function within a segment is the same as in the Ueno-Ohori-Otomo collector. In particular, no mutex or no memory barrier is required.
- GetSegment: If the state is either SYNC1-$\langle$SYNC1-LOCK$\rangle$ or SYNC2-$\langle$SYNC1-LOCK$\rangle$, it copies *AllocPtr* to *AllocPtrSnap* of *Current*. It moves *Current* to *Filled* and attempts to get one segment from *Active* or *Free* in this order. If there is no segment available then it halts until at least one segment will be available.
- BarrieredUpdate: The compiler generates the following barriered mutation code for each pointer mutation $x := y$ except for object initialization:

```
if (state is either SYNC1-⟨SYNC1-LOCK⟩,
                    SYNC2-⟨SYNC1-LOCK⟩, or
                    SYNC2-⟨SYNC2-LOCK⟩)
  remember(y);
if (state is either SYNC1-⟨SYNC1-LOCK⟩,
                    SYNC2-⟨SYNC1-LOCK⟩,
                    SYNC2-⟨SYNC2-LOCK⟩, or
                    MARK-⟨MARK-LOCK⟩)
  remember(!x);
x := y;
```

  where `remember(`y`)` and `remember(!`x`)` are the snooping and snapshot write barrier, respectively.
- CheckGC: This must be performed in loop headers, function calls, and GetSegment calls so that the mutator can detect phase changes as soon as possible. If the state is SYNC1-$\langle$ASYNC-LOCK$\rangle$, it increments HANDSHAKE and changes the state to SYNC1-$\langle$SYNC1-LOCK$\rangle$. If the state is SYNC2-$\langle$SYNC1-LOCK$\rangle$, the mutator saves *AllocPtr* to *AllocPtrSnap* of *Current*, pushes its own root set to the collector's stack, increments HANDSHAKE, and then changes the state to SYNC2-$\langle$SYNC2-LOCK$\rangle$.

- Leave: This is performed when a thread leaves the GC context. It changes its state either from SYNC1-⟨ASYNC-LOCK⟩ to SYNC1-⟨SYNC1-UNLOCK⟩, from SYNC2-⟨SYNC1-LOCK⟩ to SYNC2-⟨SYNC2-UNLOCK⟩, or from g-⟨g-LOCK⟩ to g-⟨g-UNLOCK⟩. Then, it performs the same action as CheckGC according to the original state.
- Enter: This is performed when a thread comes back to the GC context. If its state is matched with g-⟨l-UNLOCK⟩, then it changes the state to g-⟨l-LOCK⟩. Otherwise, it waits until the state becomes UNLOCK.
- Start: This is performed when a mutator thread starts. It obtains THREADLOCK, copies INITSTATE to its state, increments NTHREADS, and then releases THREADLOCK. After this, a thread typically executes Enter to start a user program.
- Exit: This is performed when a mustator thread exits. Before performing this, it must execute Leave. It obtains THREADLOCK, decrements NTHREADS, and then releases THREADLOCK.

When the collector is invoked, it performs the following sequence of actions:

1. The collector obtains THREADLOCK to fix the set of mutator threads.
2. For each thread, the collector changes its state from ASYNC-⟨ASYNC-a⟩ to SYNC1-⟨ASYNC-LOCK⟩. If a is UNLOCK, it increments HANDSHAKE and changes the state to SYNC1-⟨SYNC1-UNLOCK⟩. Wait on HANDSHAKE until it is equal to NTHREADS.
3. When HANDSHAKE becomes full, it resets HANDSHAKE. At this point, all mutators turn on all write barriers. It atomically swaps Filled with the empty list and sets the obtained segment list to $\mathbf{C}$. Then, for each thread, the collector changes its state from SYNC1-⟨SYNC1-a⟩ to SYNC2-⟨SYNC1-LOCK⟩. If a is UNLOCK, it increments HANDSHAKE, enumerates the root set of that thread, and changes the state to SYNC2-⟨SYNC2-UNLOCK⟩. Wait on HANDSHAKE until it is equal to NTHREADS.
4. When HANDSHAKE becomes full, it resets HANDSHAKE. At this point, the set of all the root sets have been pushed onto the collector's trace stack. It changes the states of all mutators from SYNC2-⟨SYNC2-a⟩ to MARK-⟨MARK-a⟩ and releases THREADLOCK. Then, it clears all the bitmaps and saves all the allocation pointers of the segments in $\mathbf{C}$ and performs marking just as the algorithm shown in Section 3.1.
5. When the marking completes, it changes the states of all mutator threads from MARK-⟨MARK-a⟩ to ASYNC-⟨ASYNC-a⟩ and moves each segment in $\mathbf{C}$ to appropriate lists in the same way as the algorithm in Section 3.1.

4. Implementation and Performance Issues

Our goal now is to faithfully implement this abstract concurrent snapshot model, which requires carefully developing several mechanisms. Furthermore, we want the resulting implementation to realize the abstract model with the minimal possible overhead so that we can achieve our goal of having a fully concurrent garbage collection that is almost as efficient as the sequential one when it is run with a single thread program.

We have implemented most all the features presented in the previous section in a Standard ML compiler. We choose the SML# compiler, which compiles the full set of Standard ML with native multithread support into x86_64 native code. This section describes the mechanism and performance issues we have considered in some details. The SML# version 3.0.0 or above includes the implementation and is available from Tohoku University as open-source software under a BSD-style license.

4.1 Collector Invocation

In the abstract model, we have assumed that the collector is invoked at some appropriate timing, for which we need to develop some policy and mechanism that guarantees that the collector is invoked just sufficiently often. For this purpose, we have implemented the following mechanism. The collector is woken up every time some of the mutators attempt to move a segment to Filled. Since garbage collection is only needed when the number of free blocks becomes few, this guarantees that the collector is woken up sufficiently often. When woken up, the collector estimates the current memory demand and decides whether collection is necessary or not. We have been searching for the best heuristics for this decision. The following is the current heuristics that works reasonably well:

1. If the number of segments in Filled is greater than the separately maintained threshold N, then the collector starts a new garbage collection cycle.
2. Every time when the segment reclamation is finished, the threshold N for the next collector invocation is computed as follows:
$$N = (\mathit{numSegments} + \mathit{numFilled})/2$$
where $\mathit{numSegments}$ is the total number of segments allocated in the system and $\mathit{numFilled}$ is the number of segments that the collector moved from $\mathbf{C}$ to Filled.

4.2 Tracing Objects without Marking

Another point to be detailed is how the collector traces objects. In the standard mark-and-sweep algorithm, the collector traces only the pointer to unmarked objects so that it can avoid tracing the same object twice. In contrast, in our algorithm, the collector must trace an object even if it is marked since $\mathbf{H}(t_0)$ includes objects the mark bit of which has not been cleared. In addition, the remembered set $\mathbf{R}$ and snoop set $\mathbf{S}$ must be maintained for mutators to pass pointers to the collector during tracing. Memory contention between mutators and the collector may occur frequently on the manipulation of $\mathbf{R}$ and $\mathbf{S}$.

In order to trace objects efficiently, our implementation uses the following data structures for object tracing:

- TMPROOT: the thread-local list of objects for a mutator to insert objects during its root set scan.
- ROOTS: the list of objects to accumulate the result of root set enumeration.
- BARRIER: the list of objects to which the write barriers stores object pointers. This is the implementation of $\mathbf{R}$ and $\mathbf{S}$.
- TRACE: the list of objects for the collector to trace.

The following data structure is added to each segment:

- $\mathit{TraceBitmap}$: the bitmap allocated in each segment to determine the set of objects the collector has traced.

$\mathit{TraceBitmap}$ and TRACE are used only by the collector. ROOTS and BARRIER is shared among all the mutators and collector. TMPROOT is temporally allocated for each thread in SYNC2 phase.

These lists are implemented as a linear linked list of objects. To form the linked list, we allocate an extra pointer slot for each allocation block in a segment. When a mutator or collector adds an object to one of the above list, it updates the extra slot of the object from null pointer to non-null pointer by compare-and-swap atomic operation. If the extra slot already holds a non-null pointer, i.e., the compare-and-swap operation failed, the addition is canceled since the object is already included in a list. This mechanism avoids duplicate objects in the remembered set and snoop set.

By using these data structures, object tracing is performed as follows:

1. When a mutator enumerates its root set, it sets up its own TMPROOT and adds objects to TMPROOT.
2. After the enumeration is finished, the mutator concatenates TMPROOT to ROOTS.
3. The write barrier code adds objects to BARRIER.
4. At the beginning of tracing, the collector clears $TraceBitmap$ of all segments and moves all objects in ROOTS to TRACE.
5. The collector uses TRACE as a tracing stack. When TRACE becomes empty, it atomically swaps BARRIER with empty list and, if some objects are obtained, it adds them to TRACE and continues tracing. If both TRACE and BARRIER are empty, the tracing is complete.

4.3 Automatic Heap Size Adjustment

We implement a mechanism to automatically adjust the sizes of sub-heaps by estimating the mutators' size-dependent memory demand. Since the non-moving Ueno-Ohori-Otomo collector does not perform compaction, this mechanism is important in achieving better memory usage and performance.

Our current strategy to adjust the heap size is as follows:

1. The ratio of the initial heap size against the entire heap space is given as a tuning parameter. We currently choose 80% as the ratio. When the program starts, the collector allocates segments and puts 80% of the segments to $Free$.
2. The system manages the following two counters for each sub-heap H_i: $NumBlocks_i$ for the number of all blocks in H_i, and $NumFreeBlocks_i$ for the number of unmarked blocks in $Active_i$. By using these counters, the collector computes the minimum number of segments additionally required to keep the percentage of $NumFreeBlocks_i$ against $NumBlocks_i$ greater than a constant threshold for each H_i. We currently choose 50% as the threshold. Every time the segment reclamation is finished, the collector performs this computation and allocates free segments in $Free$ if $Free$ is not large enough.
3. To prevent a sub-heap from growing too large, each sub-heap H_i has a 1-bit flag $DoNotExtend_i$ indicating permission to allocate a free segment to the sub-heap. If $DoNotExtend_i$ is turned on, a mutator does not add any segment to $Active_i$ of H_i. Every time the segment reclamation is finished, the collector checks whether or not the size of a sub-heap is larger than a given threshold. If H_i is too large, the collector turns on its $DoNotExtend_i$ until the next collection cycle is complete.

4.4 Minimizing Allocation Overhead

For functional programs, it is particularly important to minimize allocation overhead. The major causes of the overheads are memory contention among mutators and between a mutator and the collector, and the costs for multithread management.

Concurrent garbage collection in general may incur memory contention overhead in accessing shared data structures, such as a free list, an object state, a trace stack, and handshaking data. The combination of the Ueno-Ohori-Otomo collector and our concurrent snapshot abstraction provides a suitable framework to minimize these overhead.

In our method, most allocation requests are served within a segment owed by each mutator thread without any mutual exclusion. Memory contention occurs only when the current segment becomes full and a new segment is requested. In addition, the mutator never needs to manipulate global object state such as "color" managed by the collector with color-based graph tracing. The structure of the Ueno-Ohori-Otomo collector enables checking whether or not an object o belongs to $\mathbf{H}(t)$ by the mark bit of o and by the comparison of the position of o to $AllocPtrSnap$. Both of the two data structures do not cause any memory contention. As we shall show, our benchmark results substantiate this claim that allocation is done almost as efficient as the sequential collection.

4.5 Multithread Support Overhead

To support concurrent multithread programs running on multicore CPUs, it is necessary to maintain a thread-local context to access thread-local resources. $Current$ is the important example of a thread-local resource that is frequently accessed. To implement thread-local context, we use OS-supplied thread-local storage. Accessing this storage requires some overhead.

Another overhead due to multithread support is to insert code fragments that call CheckGC. Our implementation executes the status check code at the beginning of function calls and execution of GetSegment function call. Among them, checks at the entry of functions may incur non-negligible overhead.

Although we have carefully coded to minimize these, these two are non-negligible overhead in our system and perhaps in any system that realizes fully concurrent garbage collection. As we shall demonstrate later, our performance evaluation shows that this overhead is reasonable, and that these overheads constitute almost all of the overhead in our system.

5. Performance Evaluation

Our general goal is to develop a concurrent garbage collector having the following features:

1. its overhead should be low enough to be used as an alternative to a sequential collector for ordinary single thread programs,
2. it should scale to multithread programs on multicore systems, and
3. it should open up the possibility for real-time applications on multicore systems.

This section reports our present evaluation for the above three points. The second and third point requires a certain amount of consideration, and we leave detailed evaluation of those points for future investigation.

Evaluation has been performed on a machine equipped with two 8-core Intel Xeon E2690 2.90 GHz processors forming a NUMA (non-uniform memory access) architecture with 64 GB main memory. The operating system we used is Debian GNU/Linux. Experiments were performed multiple times with the highest OS process scheduling priority when the system is not heavily loaded.

5.1 Benchmark Programs

To evaluate overheads of multithread and concurrency support, we use benchmark programs distributed with the SML# compiler source code. They are publicly available benchmark programs for Standard ML with some modification to be adapted to the SML# compiler. We omit benchmarks such as `fib`, `tak`, and `ntakl` whose memory usage is trivially small. In addition to the micro benchmarks, we also used the SML# compiler as our benchmark program. The SML# compiler consists of approximately 160 thousand lines of Standard ML (SML#) codes and 16 thousand lines of C codes. This is a typical allocation-intensive functional program that performs a series of term transformations and therefore serves as a benchmark for a practical application. This is also a large practical open source system written in Standard ML. To use the compiler as a benchmark program, we choose the `MatchCompile` module (1579 lines), which is part of the SML# source code, as the input data and measure the time to compile it to an object file. We refer to the benchmark as smlsharp.

With regard to scalability, we could not find appropriate publicly available benchmarks for Standard ML, partly due to the fact that concurrent ML programs that use native threads on multicore CPUs are not widely available. Therefore, we have written the following three small multithread programs:

- matrix. This benchmark computes a multiplication of two 840×840 matrices with multiple threads. Each matrix is implemented as a nested array of double precision floating-point numbers. The destination matrix is divided into disjoint portions of equal size. Each thread computes the result for one of the portions and destructively updates the destination array with the result. This benchmark does not cause any memory allocation during its main iteration; therefore, the collector does nothing in this benchmark. This is a benchmark to test whether or not the threads are really concurrent.
- matrix_cps. This is a variant of matrix for the garbage collection performance test. The main iteration is written in a continuation-passing style so that mutators generate a number of short-lived closures.
- nqueen. This is a simple solver of the 14-queen problem. The search space is divided into small portions that form a task queue. Each thread repeatedly obtains a space from the queue until the queue becomes empty. The chess board is represented by a record of unsigned integers so that each thread allocates short-lived objects as intermediate data structures, while perfectly-tuned implementation of this algorithm in an imperative language would not allocate intermediate data.

We note that both benchmarks are written in Standard ML with the OS supplied POSIX thread APIs bound through the foreign function interface of SML#. In particular, user threads directly correspond to POSIX native threads on a multicore processor. For example, a parallel map can be roughly implemented by the following code:

```
fun pmap f l =
    map pthread_join (map (pthread_create f) l)
```

where `pthread_create` and `pthread_join` are SML# direct bindings of C functions of the same name.

5.2 Measurement and Settings

To evaluate our fully concurrent collection method, we prepared the following collector implementations for performance comparisons:

- **sequential**. This is the plain, non-generational, and sequential version of the Ueno-Ohori-Otomo collector. This does not support multithread. This collector has been carefully tuned and therefore serves as a standard of the efficient collector for ordinary single thread programs.
- **stop-the-world**. This is the stop-the-world collector with multithread support described in Section 3.2.
- **concurrent**. This is our implementation of the fully concurrent collector proposed in this paper.

In addition, for comparison purposes, we implemented a variant of **sequential** with the code to check the thread state, described in Section 3.7. Such checks are pure overhead for **sequential**; we made this version to single out the cost of the status flag checks, which is the major overhead of multithread support as mentioned above. We call this variant as **sequential-with-check**.

benchmark	min	sequential		stop-the-world		concurrent	
		exec	pause	exec	pause	exec	pause
barnes_hut2	1	0.31	0.25	0.33	0.25	0.40	0.18
boyer	4	0.44	2.62	0.45	2.69	0.44	0.97
count_graphs	1	8.33	0.12	8.64	0.12	9.24	0.00
cpstak	1	0.81	0.06	0.85	0.07	0.95	0.00
diviter	1	1.29	0.09	1.31	0.10	1.51	0.00
divrec	1	1.34	0.09	1.35	0.10	1.51	0.00
fft	1	0.14	0.08	0.14	0.08	0.14	0.00
gcbench	60	0.96	24.58	0.97	24.73	0.94	28.30
knuth_bendix	8	2.95	5.51	3.01	5.53	3.04	0.00
lexgen	5	0.28	2.39	0.30	2.31	0.33	2.67
life	1	0.17	0.15	0.18	0.16	0.19	0.00
logic	1	0.92	0.54	0.95	0.57	1.17	0.32
mandelbrot	1	12.60	0.06	12.73	0.06	14.58	0.00
matrix_cps	1	32.21	1.55	30.79	1.59	39.06	0.18
mlyacc	5	0.16	2.39	0.16	2.37	0.19	2.94
nqueen	1	2.94	0.08	3.12	0.79	3.33	0.00
nucleic	1	0.10	0.14	0.10	0.15	0.11	0.00
perm9	168	3.79	320.61	3.80	320.03	6.35	370.81
puzzle	1	1.10	0.06	1.09	0.07	1.15	0.00
ratio_regions	12	35.85	9.49	36.38	9.48	38.62	8.87
ray	1	0.25	0.09	0.27	0.09	0.29	0.00
smlsharp	266	5.71	222.98	5.71	222.62	5.71	295.13
tsp	6	0.43	7.29	0.43	7.42	0.51	8.18
vliw	2	0.77	1.26	0.78	1.20	1.52	3.08

(min: minimum heap size in MB. exec: average total execution time with each garbage collection method in seconds. pause: maximum pause time in milliseconds.

Table 1. Overhead against single thread programs

5.3 Cost of Native Thread Support and Concurrency

To evaluate the cost of native thread support and concurrent collection, we measured the performance of **concurrent**, **stop-the-world**, **sequential-with-check**, and **sequential** against the sequential Standard ML benchmarks. The difference between **sequential** and **sequential-with-check** is the status check overhead to support multithread programs on multicore systems. The difference between **sequential-with-check** and **stop-the-world** is the overhead of executing the collector in a different thread. Finally, the difference between **stop-the-world** and **concurrent** reflects overhead and speed up due to concurrent collection.

For each benchmark, we measured the minimum heap size in MB to run the benchmark with **sequential**. By using this heap size as a reference heap size, we measured the total execution time of every combination of a collector and benchmark with varying heap size. Figure 2 shows a summary of the evaluation results with 3 different heap sizes and their average. Table 1 shows some details of these results. These results establish the following:

1. The total overhead of our fully concurrent collector is reasonably low. It is roughly 12% on average.
2. The status check overhead is negligible.

Among the benchmarks, our concurrent collector anomalously shows proof performance against perm9 and vliw. We investigated the cases by taking various timing and profiling to find out the following. The slow execution time in perm9 and vliw is due to the mutator's occasional waits for free segment. perm9 produces large amounts of live objects very quickly, and about 51% of the total time is wait times, which is exceptionally high. Partial collection in our concurrent collection sometimes fails to catch up with the mutator's speed of allocation, causing a mutator to wait. vliw with 4 MB heap size ($2 \times$ minimum heap) also consumes about 44% of its total time for free segment wait. However, this wait time becomes much smaller with larger heap sizes. This indicates that the 4 MB

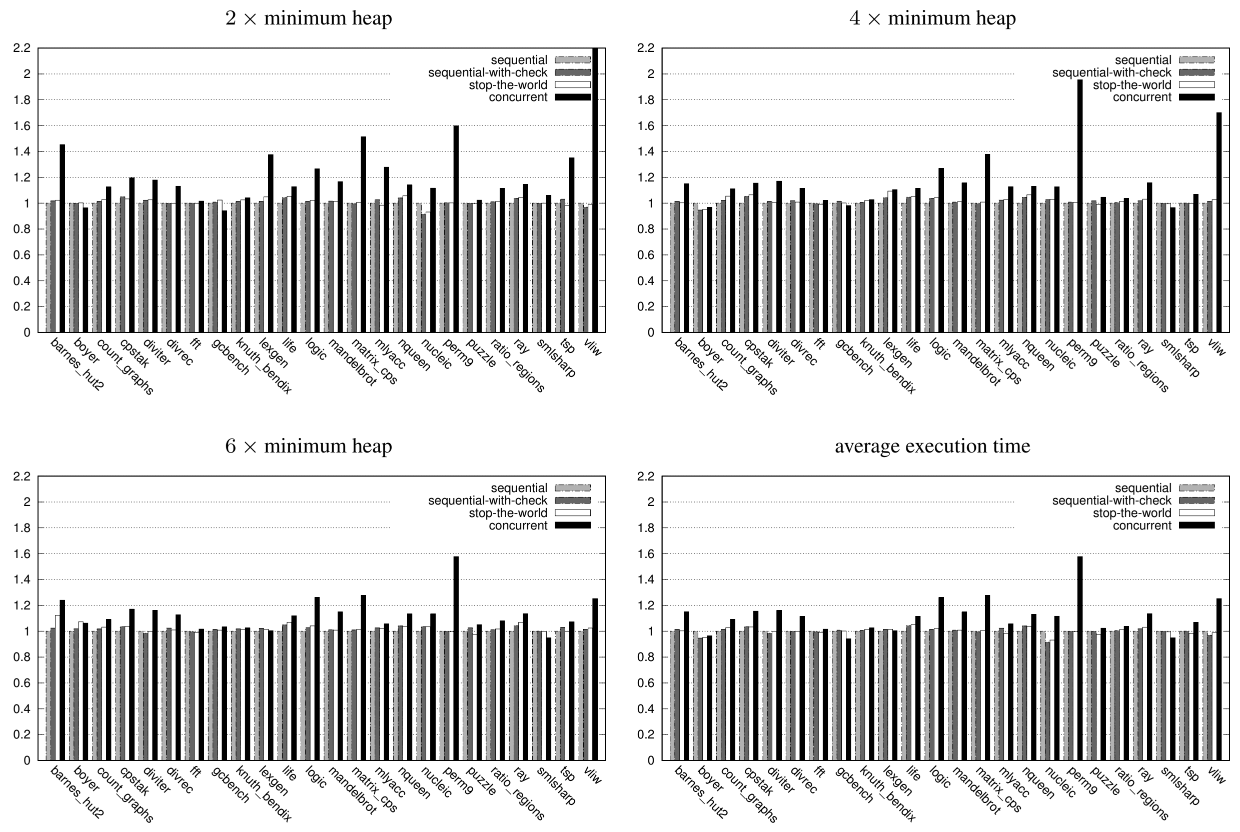

Figure 2. Overhead of multithread support against the original version

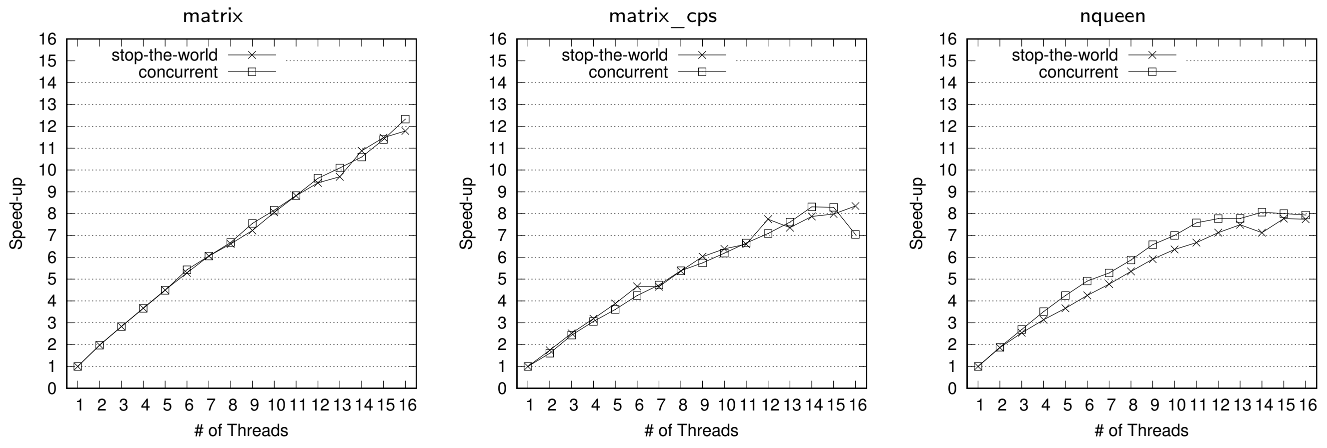

Figure 3. Scalability for multithread benchmarks

heap size is too small to run vliw with our concurrent collector and the current segment scheduling.

From these results, we conclude that the overhead due to concurrent collection is small enough to be acceptable. In particular, our concurrent collection performs almost as efficiently as a sequential collector performs against typical functional programs that do not perform intensive mutation.

Quantitative evaluation of these results with other methods is rather difficult owing to a lack of published results on these costs in functional languages. The implementation of the Doligez-Leroy-Gonthier collector [9, 10] does not report any relevant results to compare with. The only published results on these overhead in a functional language compiler we are aware of are those of the parallel and concurrent collector implemented for a Standard ML compiler [5], which shows 30% to 80% slow-down against sequential collection. Since their collection method is significantly different from ours, accurate comparison is difficult. Nonetheless, we would regard that our result of 12% overhead on average as the best possible result so far achieved for functional languages to date.

5.4 Scalability on Multicore Systems

The next evaluation is the overall performance and scalability on multicore systems. Figure 3 shows total execution times with varied thread numbers for each of the three concurrent benchmarks. Table 2 shows some details of these results.

In the range of 1 to 8 threads, we conclude that our system exhibits the expected speed-ups for the typical parallelizable problems. We also conclude that the scalability of **concurrent** is comparable, or slightly better in nqueen, than that of **stop-the-world**. We note that the speed-up of our benchmark programs is up to 8 times in matrix_cps and nqueen whereas the machine we used has 16 cores in total. As seen in the result of matrix, if there is no memory allocation, the speed-up is up to around 12 times. At this moment, we are unable to identify the reason of this limitation. A possible reason we guess is that this is due to the NUMA architecture. In our implementation, only the collector allocates heap memory by mmap system call; therefore, if a mutator thread runs on a different CPU than the one on which the collector runs, the mutator would suffer performance penalties associated with remote memory accesses. Detailed analysis of the performance of our collector on a NUMA machine and implementation improvement based on it are left to further investigation.

We are aware that these benchmarks on scalability are rather small, only using a small number of cores and threads. Benchmark tests with much larger numbers of cores and threads should only be meaningful when the collectors produce enough amount of allocable memory to keep up with the consumption of all the threads. Our current implementation is limited to one collector, which effectively bounds the number of threads. We believe that it is not difficult to extend our collection method to allow parallel collections by multiple collector threads. We would like to perform more substantial scalability benchmarks after this extension.

5.5 Maximum Pause Time

In these two sets of benchmark programs, we have also measured the maximum pause time due to garbage collection, including root set scan. The results are shown in "pause" columns of Table 1 and 2. In these tables, the maximum pause time for **sequential** is equivalent to the maximum duration of a garbage collection cycle, and that for **stop-the-world** is the combination of garbage collection time and stop-the-world time. The results show that our concurrent collector significantly reduces the maximum pause time.

We note that the maximum pause time depends on the total memory demand and the collection speed, as exhibited by perm9 and smlsharp. As we have analyzed above, perm9 shows exceptionally

threads	benchmark	heap	stop-the-world		concurrent	
			exec	pause	exec	pause
1	matrix	32	11.38	1.44	11.32	0.00
	matrix_cps	32	25.66	1.59	21.62	0.00
	nqueen	32	3.19	0.79	3.31	0.00
2	matrix	32	5.75	1.45	5.74	0.00
	matrix_cps	32	14.71	1.50	13.48	0.00
	nqueen	32	1.71	0.61	1.76	0.00
4	matrix	32	3.10	1.26	3.09	0.00
	matrix_cps	32	8.04	1.58	7.06	0.00
	nqueen	32	1.02	0.72	0.94	0.00
8	matrix	32	1.72	1.43	1.69	0.00
	matrix_cps	32	4.78	1.74	4.01	0.00
	nqueen	32	0.60	0.86	0.56	0.00
16	matrix	32	0.97	1.44	0.92	0.00
	matrix_cps	32	3.07	1.58	3.07	0.56
	nqueen	32	0.41	0.96	0.42	0.00

(threads: number of threads. heap: heap size in MB. exec: execution time in seconds. pause: maximum pause time in milliseconds.

Table 2. Scalability detail for multithread benchmarks

high memory demand, for which our concurrent collector sometimes fail to catch up, resulting in wait for segment reclamation. This is the case of long maximum pause time of perm9 benchmark. smlsharp similarly shows high memory demand. In addition, smlsharp changes allocation mode during its execution because of different memory demand in each compilation phase. This causes frequent invocation of garbage collection and sub-heap exhaustion, whereas the total execution time with **concurrent** is faster than **sequential**.

6. Related Works

In the Introduction, we already discussed and compared with some related work [4, 5, 9, 10, 13, 18]. In addition to these, there are several relevant ones. In [2], the authors suggested that they have developed and implemented a collector for the parallel functional language Manticore, which is a hybrid system of the Doligez-Leroy-Gonthier collector and a local collector reported in [1]. However, from their brief description of the system and also their statement that "the global collection is a parallel stop-the-world collector," their collector does not seem to address unobtrusiveness in the sense of [9, 10]. Marlow and Peyton Jones [15] reported a parallel collection method for a full scale Haskell compiler and provided detailed performance results. This work demonstrated that parallel collection is a feasible approach when more than one core is available in a practical language compiler. However, their system is the stop-the-world collector and concurrent collection is not addressed.

Various concurrent garbage collection methods including [3, 6, 7, 11, 16, 17] have been proposed and implemented for Java, especially for server use, which requires short pause time. These systems provide various insight into our design and implementation of concurrent garbage collection for functional languages. Most of these systems have been developed on Java Virtual Machine to support Java code. These have rather different properties than functional programs which have high demand of large amounts of short lived data. Our work specifically targets those functional programs. For this purpose, we paid particular attention to very fast allocation.

7. Conclusions

We have developed a fully concurrent garbage collection method suitable for functional languages running on a multicore processor. It is a concurrent extension of the Ueno-Ohori-Otomo non-moving collector based on Yuasa's *snapshot-at-the-beginning* abstraction.

We have constructed a formal model for concurrent snapshot abstraction, and have established its correctness. It is an unobtrusive collection in the sense of [9, 10]. The only critical sections between mutators and the collector is the code to obtain a segment from a global free list of segments, and the portion of write barrier code that inserts a pointer into a global list. The resulting collector has the desired properties for memory demanding functional programs. The mutator's stop time is very short. Allocation in ordinary cases is as efficient as the underlying sequential collection. Different from concurrent collectors based on tricolor marking, it does not require repeated scans of the heap and the collection time is bounded by the amount of allocated blocks at the time when the collection starts. Our benchmark results show that the proposed collection method is an efficient and scalable one suitable for functional programs on multicore systems, and that it can serve as a viable alternative to currently used sequential collection in a practical functional language. The presented collection method has been fully implemented in the SML# compiler, which compiles the full set of Standard ML language (with seamless interoperability with C) into x86_64 native code. According to our benchmark results, the proposed method exhibits the expected performance. For multithread parallel programs, the proposed system shows expected scalability and very short pause time. For single-thread ordinary ML benchmark programs, the total overhead of our concurrent collector is reasonably low; it is roughly 12% of the carefully tuned sequential collector. From these, we can conclude that the proposed method is an efficient and scalable one suitable for functional programs on multicore systems, and that it can serve as a viable alternative to currently used sequential collector in a practical functional language.

Although the concurrent snapshot abstraction for a non-moving collector has the desired properties for functional programs and our performance evaluation is very promising, there is room for improvement and a number of important and interesting issues remain to be investigated. The most important improvement is parallel collection. The reported method has only one collector thread. However, we believe that it is relatively straightforward to extend our method so that collection can be performed in parallel. In our segment based heap space organization, any subset of filled segment can be independently collected. We can then develop a parallel and concurrent non-obtrusive collector using the technique of work-stealing queues [12] similarly to the parallel collector for Haskell [14, 15].

Another important issue is further evaluation of the scalability and real-time performance of our concurrent collector. This should be conducted to reveal more detailed characteristics of the concurrent collector, such as applicability to NUMA architectures. The parallel benchmark suite should also be extended using much more widely known parallel algorithms.

Acknowledgments

The first and the second authors have been partially supported by the JSPS KAKENHI Grant Number 25280019 under the title "Foundational research on implementation methods for the ML-style polymorphic programming language SML#." The second author has also been partially supported by the JSPS KAKENHI Grant Number 15K15964 under the title "A systematic approach for developing practical programming languages."

References

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, 1989.

[2] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pp. 51–57, 2011.

[3] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 269–281, 2003.

[4] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 104–117, 1999.

[5] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 125–136, 2001.

[6] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pp. 46–56, 2005.

[7] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pp. 37–48, 2004.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21:966–975, 1978.

[9] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 70–83, 1994.

[10] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 113–123, 1993.

[11] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 274–284, 2000.

[12] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, pp. 21–21, 2001.

[13] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[14] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, pp. 11–20, 2008.

[15] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the international symposium on Memory management*, pp. 21–32, 2011.

[16] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 129–140, 2002.

[17] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pp. 159–172, 2007.

[18] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pp. 146–159, 2010.

[19] SML# project. http://www.pllab.riec.tohoku.ac.jp/smlsharp/.

[20] K. Ueno, A. Ohori, and T. Otomo. An efficient non-moving garbage collector for functional languages. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pp. 196–208, 2011.

[21] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.