A Lock-Free, Concurrent, and Incremental Stack Scanning for Garbage Collectors

Gabriel Kliot
Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel *
gabik@cs.technion.ac.il

Erez Petrank
Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel †
erez@cs.technion.ac.il

Bjarne Steensgaard
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
Bjarne.Steensgaard@microsoft.com

Abstract

Two major efficiency parameters for garbage collectors are the throughput overheads and the pause times that they introduce. Highly responsive systems need to use collectors with as short as possible pause times. Pause lengths have decreased significantly during the years, especially through the use of concurrent garbage collectors. For modern concurrent collectors, the longest pause is typically created by the need to atomically scan the runtime stack. All practical concurrent collectors that we are aware of must obtain a snapshot of the pointers on each thread's runtime stack, in order to reclaim objects correctly. To further reduce the length of the collector pauses, incremental stack scans were proposed. However, previous such methods employ locks to stop the mutator from accessing a stack frame while it is being scanned. Thus, these methods introduce a potential long and unpredictable pauses for a mutator thread. In this work we propose the first concurrent, incremental, and lock-free stack scanning for garbage collectors, allowing high responsiveness and support for programs that employ fine-synchronization to avoid locks. Our solution can be employed by all concurrent collectors that we are aware of, it is lock-free, it imposes a negligible overhead on the program execution, and it supports the special in-stack references existing in languages like C#.

Categories and Subject Descriptors D.1.5 [*Object-oriented Programming*]: Memory Management; D.3.3 [*Language Constructs and Features*]: Dynamic storage management; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage Collection

General Terms Algorithms, Design, Performance, Reliability

Keywords Stack scanning, Lock-free data structures, Incremental and Concurrent garbage collection

* Work done while the author was an intern at Microsoft Research.

† Work done while the author was on sabbatical leave at Microsoft Research

VEE'09, March 11–13, 2009, Washington, DC, USA.

1. Introduction

Garbage collection is widely acknowledged to speed up software development while increasing security and reliability. Garbage-collection has been incorporated into modern popular languages such as C# or Java. However, garbage collectors may create pauses in the execution of the application, and introduce an overhead that reduces its efficiency. In this work, we concentrate on reducing the pause times. We deal with the main disruptive element for concurrent collectors and propose a method that reduces pauses to the microsecond level, while hardly affecting the efficiency and without using locks.

When garbage collection was first proposed and implemented, the application was halted during garbage collection execution, creating long pauses. In order to mitigate this problem, especially on modern parallel platforms, concurrent garbage collectors were proposed [26, 27, 12, 13, 8, 23, 6]. Concurrent garbage collectors run concurrently with the application and only stop it for a short synchronization phase in the beginning or end of the collection. On-the-fly collectors are special concurrent collectors that never need to stop all threads simultaneously. They stop the application one thread at a time typically for scanning the thread runtime stack [15, 14, 16, 17, 20, 3].

Thus, the major remaining pause bottleneck for modern on-the-fly collectors is the stack scanning of a single thread. These collectors require an atomic snapshot view of the stack in order to execute correctly. In this paper, we propose an incremental, concurrent, and lock-free stack scanning that is adequate for all known on-the-fly collectors. Incremental means that the stack is not scanned at once, but one frame at a time. Concurrent means that the collector may scan the frames while the mutator thread is executing. This requires some synchronization between the collector and mutator. However, lock-freedom means that this synchronization is ensured at a fine-grained level, without locks, and with progress guaranteed. In spite of the incremental nature of the scan, the output of the stack scanning procedure equals the one that would have been obtained by an atomic snapshot of the stack. The main idea is to allow both the collector and the mutator to scan the stack simultaneously. The mutator thread is required to scan only a single frame at a time, only if it is about to use or modify it, and only if the collector was not fast enough to scan it earlier. This happens upon exit from a method, and is executed with an efficient *return barrier*. It is expected that much of the scanning work will be executed by the collector in parallel, off-loading management work from the application threads.

On top of being concurrent, incremental and lock-free, our algorithm supports interaction between managed and unmanaged code. In particular, it is possible to handle a stack that contains frames of the managed language and frames that represent system

calls written using some low level language. It is also possible to handle threads that are currently in the midst of performing system calls and are not responding to the collector's handshake requests[1].

Our basic design is extended to support intra-stack pointers as well as moving garbage collectors. Handling moving collectors requires supporting a stack scan that may need to update pointers to objects that have been relocated. In addition, we describe how we deal with callee saved registers conventions.

Short pause times are needed most when high responsiveness is required. Particularly in such cases, it is important that the synchronization is fine-grained and locks are not taken. A program task should not be delayed just because the collector acquires a lock and is then preempted while holding it. Therefore, in order to synchronize the concurrent actions of the collector and the mutator thread in our method. The algorithm employs fine-grained synchronization via a compare-and-swap (CAS) instruction which is common to stock hardware. We do not employ the special DCAS instruction, which typically simplifies synchronization but is not commonly found on standard platforms.

We have implemented our algorithm on top of the BARTOK compiler and runtime for C#. Measurements demonstrate pause times at the level of a small number of microsecond and a negligible throughput decrease.

Organization In Section 2 we survey some basic facts about on-the-fly collectors and stack scanning for them. In Section 3 we describe the stack scanning method. In Section 4 we describe an extension for moving collectors and in Section 4.4 we describe an extension to support for the intra-stack pointers of C#. Implementation and measurements are reported in Section 5 and related work is described in Section 6. We conclude in Section 7.

2. On-the-fly Collection and Stack Scanning

Tracing garbage collectors trace the set of objects that are reachable by the program, and then reclaim all objects not in this set. Mark-sweep collectors mark the reachable objects and then sweep the heap to reclaim all unmarked objects, whereas copying collectors move all reachable objects to a reserved space, and then reclaim the entire original space, which then becomes the reserved space for the next collection. Reachability of objects relates to a set of pointers that are directly accessible by the program, known as roots. This set typically contains the runtime stack, the registers, and global variables. The major component in this set and the one that typically requires most time to determine is the set of pointers on the stack. Stack scanning is not only required for tracing collectors. Reference-counting collectors maintain for each object a count of pointers that reference it. Most reference-counting collectors employ the *deferred* reference counting method [11, 4, 20, 2, 7], which also requires a stack scan during a collection. The pause times of on-the-fly reference-counting collectors are mostly determined by the time it takes to scan the stacks. Thus, on-the-fly tracing and reference-counting collectors need an improved stack scanning method if one needs to further reduce their pause times.

Since call stacks in principle can grow substantially in size, the time it takes to scan the call stacks for roots can also grow substantially. Garbage collectors that need to support high responsiveness must therefore be able to bound the pause times caused by stack scanning in order to provide any real-time guarantees.

On-the-fly collectors typically work in *phases* that are separated by *handshakes*. For example, a collector may start with the stack scanning phase, proceed with a heap tracing phase, a sweeping phase, and finish by entering the idle phase. The mutator threads are often required to be aware of the collector phase, so that they can execute the appropriate memory barriers, or cooperate in scanning their stacks, etc. In order to avoid intrusive messages that communicate a change of phase to the mutator threads, handshakes are typically employed. To initiate a handshake, the collector raises a global flag to announce a phase change. It then waits for all mutator threads to acknowledge the change. A mutator thread checks whether a request for a handshake has been issued once in a while at its own pace. When it observes the request, the mutator thread acknowledges the phase change by raising a flag that corresponds to it. When all mutator threads acknowledge the change, the collector may proceed with performing the phase tasks. On a response to a handshake request, the mutator thread is sometimes required to perform some action. In particular, one of the handshakes is typically used to request all mutator threads to scan their stacks.

An important requirement that on-the-fly collectors make of their stack scanning procedure is that it provides an atomic snapshot of the stack. Namely, during the scan, the mutator thread is not allowed to change the pointers on the stack. This seems to require that the thread is halted during the scan. However, several researchers have attempted to mitigate this requirement by letting the mutator thread scan frames only before it is about to modify them and delay the scanning of other frames to a later time [5, 9, 28]. The basic idea that allows breaking the stack scanning into small increments, is that recording the snapshot can be delayed for data that is not being modified. Thus, as the thread can only change the pointers on the most recent stack frame, the scanning of the other frames can be delayed until the thread finishes executing the current method.

Previous work had the following shortcomings. Baker [5] required the use of memory barriers on stack access, which impose a penalty on the performance. Cheng and Blelloch [9] used a lock to coordinate the work of the program and the collector [9], which does not support lock-free programs and may cause an unexpected pause if a lock holder is preempted. Yuasa et al. [28] do not allow the collector to work concurrently with the collector, so the collector is halted until the mutator finishes the stack scan using a work-based scheduling. This avoids synchronization but reduces concurrency and creates a reduced MMU because of the work-based scheduling. In this work we provide a concurrent incremental stack scanning method that does not sacrifise efficiency or lock-freedom.

The call stack may interleave program frames (of managed code) and frames that represent native or system calls (that usually executes unmanaged code). We assume that no live object in the program heap is reachable only from frames of unmanaged code, and so we may skip tracing such frames. With moving collectors, we assume that objects that are directly pointed to by these frames are pinned and cannot be moved.[2] This is guaranteed for C#. Finally, we assume that there is enough known structure in the frames of the unmanaged code so that it is possible to jump over them and scan only frames of managed code. In our design, if a mutator thread is running unmanaged code (e.g., performing some system call), the collector is allowed to acknowledge the handshake on the mutator's behalf and scan the call stack for it. In this case, the mutator must cooperate upon returning to the managed code. However, unlike prior designs, the mutator is not blocked until the collector finishes the stack scanning.

Stack scanning typically requires compiler support. The compiler produces stack maps that indicate which slots in each stack frame hold pointers at a given code-line execution. To find the appropriate stack map to use, the scanning procedure must find the call site address, which in turn requires finding the beginning of the relevant stack frame.

[1] Handshakes are a standard form of communication between the program and an on-the-fly collector and they will be defined in Section 2.

[2] Pinned objects are objects that are especially marked unmoveable for a moving collector.

3. The Stack Scanning Algorithm

The basic idea is that when stack scanning is required, it is enough to scan only the frame that the mutator is currently running, and the rest of the stack may be scanned later. If the mutator is not modifying older frames, taking a snapshot of the pointers on the stack can be delayed, while the mutator is able to continue running concurrently. The time in which this newest frame is atomically scanned, is the time at which the stack snapshot should be taken. All values in the stack existing at this time will be (later) recorded into the resulting stack-scan output. This process may be thought of as an analogue of the snapshot-at-the-beginning technique used with concurrent garbage collection. However, a snapshot-at-the-beginning concurrent collector typically assumes that the stack is scanned atomically, when the program is stopped. We allow that initial atomic phase to be broken into several shorter incremental atomic sections in the execution.

The simplest scenario is that the mutator scans its newest frame, and then continues running using that frame (and maybe producing newer frames by making calls to more methods), while the collector concurrently scans all older frames. The pause for the mutator is limited in this case to a single stack frame scan. Furthermore, the size of the largest possible stack frame can be bounded statically. However, this simple case does not always hold, as the mutator may exit its current method and move to an older stack frame, while the collector is still scanning its stack. One possible solution is to lock further mutator activities when that happens, until the collector finishes the stack scan. However, this may pause the mutator for the whole stack scanning time and nothing is gained.

In this work, we provide a solution that lets the collector and mutator scan frames one by one concurrently. If the mutator is slow in exiting frames, then the collector scans the frames one by one until the entire stack is scanned. If the mutator exits methods fast, it may arrive at a frame that has not yet been scanned by the collector. In this case, it scans that frame itself before running the code of the method that uses that frame. Namely, the mutator scans a frame only if it is about to modify a frame that the collector has not yet scanned. Naively, this method requires adding a test to all method exits, which would create a noticeable overhead. We propose to use a *return barrier* [18, 10, 28]. The return barrier adds no overhead to the execution except when scanning work (i.e., a frame scan) is required. Usually the collector will perform all the scanning work and the mutator will hardly see any execution overhead.

To avoid further pauses originating from synchronization, we design the entire cooperation between the mutator and the collector to be lock-free. Lock-freedom typically requires a special data structure whose access uses no locks and that can properly synchronize all the activity. In our algorithm, we design a special lock-free *stack-summary* data structure, denoted by *Summary-DS*. Whenever the mutator or the collector scans a stack frame, they record the result (a list of addresses referenced from the stack frame) in the *Summary-DS*. A stack summary of each stack frame is added to the *Summary-DS* only once. Eventually the *Summary-DS* contains a list of all stack pointers, that provide a snapshot of the stack and can be used by the collector as roots for a marking phase.

The entire algorithm is lock-free (and even wait-free from the mutator point of view) in the sense that a mutator thread never needs to wait for the collector. Thus, it can be used with collectors that support lock-freedom, such as [22, 21]. Two problems arise from the lock-freedom requirement. First, the collector and the mutator may concurrently update the *Summary-DS*. Thus, consistency and uniqueness of the *Summary-DS* must be preserved. Second, the collector may scan an outdated stack frame while the mutator is changing the stack content. This may happen if the mutator has already scanned that frame, but the collector has not yet noticed this fact. These issues will be addressed in the design below.

We use an important design principle that only the mutator is allowed to write to the stack. The collector may read the stack and help to gather information, but it never modifies the mutator's stack. This helps preventing races and is particularly important for the moving-collector design.

3.1 The Return Barrier Mechanism

Return barrier [18, 10, 28] is a mechanism for trapping a return from a specific method without adding a test code to the exit procedure of all methods. This mechanism overwrites the return address of the target method F on the stack with the address of the return barrier code location, and saves the overwritten return address in a predefined location (e.g., in the thread object). This change does not effect any further execution, including entry to and exit of any method except an exit from the specific instance of F whose return address on the stack is replaced. When that instance of F finishes executing and returns, the control is automatically transferred to the return barrier code. At the end of the return barrier code, the saved return address is retrieved and used to return to the appropriate code location, which is the instruction that immediately follows the call to F. Thus, after executing the return barrier method, the program resumes at the original return address.

The return barrier mechanism is used to "trap" the mutator thread just before it is about to return to a stack frame that existed in the stack when the snapshot was requested, but was not yet scanned. We employ only a single return barrier for any thread stack at any point in time. Algorithm 1 describes the return barrier pseudo code.

Algorithm 1 General structure of the return barrier function

```
return barrier function
  if the current stack frame has not yet been scanned
     Scan it and insert a summary record into the Summary-DS;
  // Place a return barrier on the next frame in the stack
  InstallNextReturnBarrier();
  return to the previously saved address.

InstallNextReturnBarrier()
  Find the next stack frame and record its return address;
  Replace the return address of that stack frame with the
   return barrier code location;
```

The return barrier mechanism must also cope with exceptions. When an exception is thrown, the stack from the throwing location to the catching location is scanned as well, by inspecting each stack frame while unwinding the exception. If a return barrier is found, then it is executed as part of the exception unwinding code and a subsequent return barrier is placed at the next stack frame.

There are a number of alternatives to using the return barrier. The most obvious one is the insertion of conditional statements guarding the use of a stack frame prior to or after the return from a method call. The return barrier mechanism is superior in two respects. First, it allows dynamically enabling the trapping only when required. Second, the return barrier we use has a minimal overhead. It is executed by the mutator once for each stack frame that exists on the stack when the stack scanning request is made and doesn't affect the execution at all otherwise. In particular, if a return barrier is placed on a specific stack frame, then until the execution returns from that method it can enter and leave any method without any overhead whatsoever. When exiting a method associated with the frame that has the barrier, it is moved to the next frame, and thus, each frame on the stack is associated with a single barrier execution. Note that many times the collector will scan the frame before the mutator and then the barrier will only execute a move of itself to the next frame without imposing any additional overhead.

3.2 Synchronizing the start of a scan

To initiate a stack scan, the collector raises a request flag for each mutator thread with a new scan number. It then waits for all mutators to acknowledge the request. Threads that are currently executing unmanaged code do not acknowledge the request. In this case, the collector starts scanning their stacks without waiting for an acknowledgement. Special care needs to be taken when such threads return from the unmanaged code. This special care is the focus of this subsection. It is crucial to note that for managed languages like C# or Java there is no inter-thread races at this point. Each thread uses its own local stack that no other thread can access. Thus, we only need to worry about the interaction of one mutator thread with the collector.

Each mutator thread may be in one of two states. We say that a thread is in a `Managed` state if it is currently executing code of the managed language. We say that a thread is in a `Dormant` state if it is currently executing in the unmanaged space. We will need to guard transitions between the different states, and we assume the compiler allows adding code to the transition so that collector-related code can execute when the mutator thread becomes dormant or when it returns to executing managed code. These barriers in the execution of a thread will be denoted `GoToUnmanagedSpace()` and `ReturnToManagedSpace()`.

Each of the two states is further partitioned into two sub-states according to whether a scan has been requested by the collector (and not yet acknowledged by the mutator). The transitions are depicted in Figure 1 and occur when the collector sets the `scanRequest` flag, the mutator resets it, and when the mutator enters or leaves unmanaged code (becomes `Dormant` or returns to managed code). We denote the transition between the dormant and managed states by `TakeManagedControl()` and `TakeDormantControl()`. The state of each mutator is signified in a thread-local variable `State`, which can be inspected by the collector and the mutator and is modified only via a `CAS` operation.

The collector initiates the scan by invoking the `CollectorStackScan()` method, described in Algorithm 2. A unique global scan number is assigned first. As will be clear later, this number is required for consistency of the *Summary-DS* because the mutator may return from a dormant state after several collections have been executed. The collector then raises a `ScanRequest` bit for each mutator. If the mutator is not `Dormant`, then the collector waits for it to acknowledge the handshake by clearing the `ScanRequest` bit and then starts scanning its stack concurrently.

The reason the collector must wait for the mutator to respond is that the mutator is modifying the stack continuously, while running in managed code. Thus, some sort of synchronization must be set up before the concurrent scan can begin. In particular, the collector and the mutator must agree on the point in time in which the snapshot is taken. This is done by letting the mutator fix the first return barrier and record an initiation information in the *Summary-DS*. Only then, the mutator acknowledges the `ScanRequest` and lets the collector join the scanning concurrently.

We still need to address the case in which the mutator is `Dormant`. In this scenario, the collector immediately proceeds to scan the mutator's stack. This can be done because the mutator is not modifying the managed stack frames at this time. But care must be taken to coordinate the return of the mutator from the dormant space. Mutator's actions related to the unmanaged space are described in Algorithm 3. Before the mutator invokes any unmanaged function call, it records the stack pointer in a thread local variable `stackPointers` (this is required for the collector to be able to start the scan without waiting for the mutator) and switches into a `Dormant` state. Upon returning to the managed space, the mutator switches back to the `Managed` state and then starts cooperating with the regular stack scanning protocol described below. When

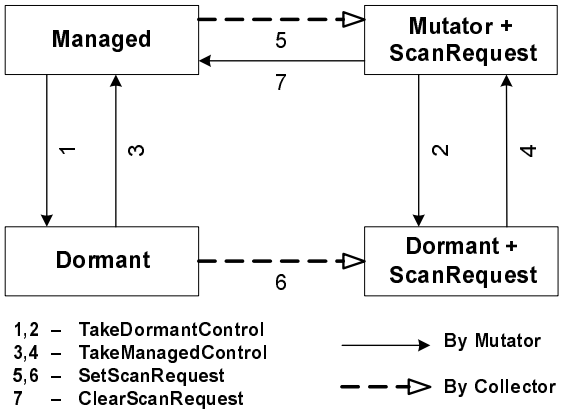

Figure 1. Mutator States Diagram

a mutator thread notices that the `ScanRequest` bit is set (when running regular managed code or upon returning from unmanaged space), it initiates the stack scanning and then clears the `ScanRequest` bit (`MutatorScanResponse()` method). Notice, however, that if the mutator is `Dormant`, the collector is able to finish scanning its stack and exit the `CollectorStackScan()` method without ever waiting for the mutator.

Algorithm 2 Initiating a Scan

```
CollectorStackScan()
  scanNumber++
  for each mutator i do
    SetScanRequest(i)
  for each mutator i do
    loop
      if InDormantState(i) then
        Summary-DS_i.ScanRootsByCollector(i, scanNumber);
        break
      else
        if HasScanRequest(i) then
          Wait(some-time);
        else
          // Mutator already started scanning - join him
          Summary-DS_i.ScanRootsByCollector(i, scanNumber);
        break

MutatorScanResponse()
  if HasScanRequest(currThread) then
    Summary-DS_currThread.ScanRootsByMutator(scanNumber);
    ClearScanRequest(currentThread);
```

Algorithm 3 Unmanaged space support

```
GoToUnManagedSpace()
  stackPointers[currentThread] = Current Stack Pointer;
  TakeDormantControl(currentThread);

ReturnToManagedSpace()
  TakeManagedControl(currentThread);
  MutatorScanResponse();
```

A simple optimization can be used when the collector finishes scanning the entire stack while there is still a pending `ScanRequest` for the mutator. In this case, there is no need for the mutator to perform any stack scanning work at all. Thus, the collector could clear the `ScanRequest` bit, if the mutator is still dormant.

This is done with a CAS operation to avoid races. Of-course the algorithm works also without clearing this flag. In particular, this optimization cannot be used with moving collectors that are discussed in Section 4. For such collectors, the mutator must be aware and take action for each stack scan, even if it does not participate in the actual scan at all.

The above handshake mechanism guarantees that each mutator thread executes the `ScanRootsByMutator()` method with the latest `scanNumber` (as long as it is not in `Dormant` state), and the collector thread will execute `ScanRootsByCollector()` method for every mutator call stack. The mutator may even skip a number of scan cycles. This does not create any problem. The ability of the collector to make progress does not depend on the mutator in this case, since the collector can finish the stack scanning itself and proceed to reclaiming garbage. This also does not violate any safety property w.r.t. the application, as the mutator takes care to not foil the scan upon returning to the managed space.

3.3 Synchronizing a frame scan

We now proceed to discuss the scan of a single frame. Since both the collector and the mutator potentially scan the same stack frame concurrently, some synchronization is needed to coordinate their actions and is achieved via the *Summary-DS* data structure, now described. For simplicity, we discuss stack frames, but the scanned stack segments may be larger and may contain more than one frame, depending on the system needs. We chose to scan one frame at a time in our system, because responsiveness was our top priority.

The *Summary-DS* is a designated data structure used to maintain records of the stack frames that were already scanned. It holds a stack-frame record for every frame of the stack, denoted `FrameRecord`, which consists of a variable length linked list of pointers (object addresses) residing in this frame on the stack. The `FrameRecord` also holds a `next` pointer pointing to the next `FrameRecord` in the chain and a `StackPointer` pointing to the end of the frame corresponding to this `FrameRecord`.

The object addresses recorded in the `FrameRecord` are part of the root set of pointers that need to be traced by a mark-sweep collector or modified by a moving collector. There is one *Summary-DS* structure per mutator stack and it is being updated cooperatively and concurrently by its corresponding mutator and the collector thread. The update is lock-free, i.e., one of the two threads is bound to make progress within a bounded number of steps.

The *Summary-DS* data structure is implemented as a linked list of frame records with a header pointer named `head` that always points to a first dummy empty record. The `next` pointer is used to link the list and the records are stored in a chronological order, so that new records are added to the tail. The `StackPointer` points to the end of the frame of the corresponding `FrameRecord` and it allows the collector to find where the next stack frame starts in the case it needs to skip a number of frames.

Both the mutator and the collector hold a local pointer to what they conceive as the last record (tail) of the list. Figure 2 depicts an example of a stack and a corresponding *Summary-DS*. In this example, the mutator has already exited three top (most recently created) stack frames and the return barrier is guarding the access to the next stack frame. The mutator's tail thus points to the third `FrameRecord`. On the other hand, the collector has already created summaries for five frames and is about to scan the sixth.

As mentioned in Algorithm 2, the mutator starts a new stack scanning by invoking the `ScanRootsByMutator()` method, which is described in Algorithm 5. This procedure starts by invoking `InitiateHeader()` on the *Summary-DS* structure (Algorithm 6). This methods starts by saving the pointer to the head frame record locally (in `tmp`). This allows testing the `number` on

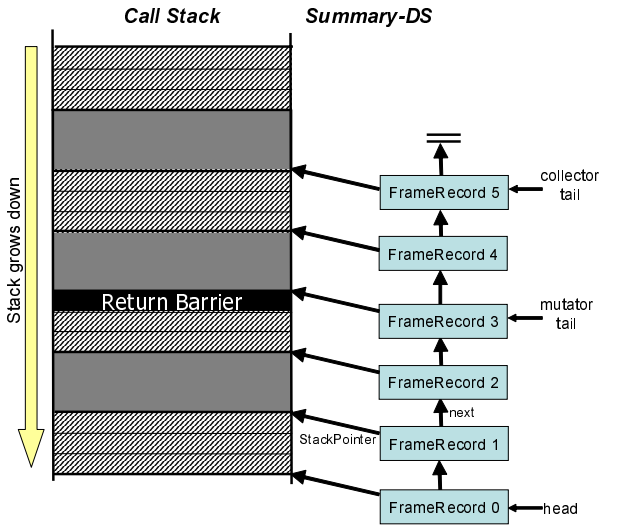

Figure 2. The call stack and the *Summary-DS*

Algorithm 4 ScanRootsByCollector(int threadId, int scanNum)

```
InitiateHeader(scanNum, stackPointers[threadId]);
collectorTail = head;
while collectorTail.addr ≠ end of the stack
    if collectorTail.next = NULL then
        FrameRecord record = ScanFrame(collectorTail.addr);
        CAS(&collectorTail.next, record, NULL);
    collectorTail = collectorTail.next;
    ApplyRecord(collectorTail, TRUE);
```

Algorithm 5 ScanRootsByMutator(int scanNumber)

```
InitiateHeader(scanNumber, stackPointers[currentThread]);
mutatorTail = head;
Eliminate the return barrier from the previous scanning cycle;
InstallNextReturnBarrier();
```

Algorithm 6 InitiateHeader(int scanNumber, int* startAddr)

```
tmp = head;
if tmp.number ≠ scanNumber then
    newHead = new FrameRecord(scanNumber, startAddr);
    CAS(&head, newHead, tmp);
```

Algorithm 7 return barrier function

```
if mutatorTail.next = NULL then
    // The current stack frame has not been scanned yet
    FrameRecord record = ScanFrame(mutatorTail.addr);
    CAS(&mutatorTail.next, record, NULL);
mutatorTail = mutatorTail.next;
ApplyRecord(mutatorTail, FALSE);
// Move the return barrier one frame up the stack;
InstallNextReturnBarrier();
return to the previously saved address.
```

Algorithm 8 ApplyRecord(FrameRecord record, bool isCollector)

```
if isCollector then
    mark and trace addresses in the record;
```

the same head record as a previous value for the later CAS operation. After initiating the header, the mutator installs a new return barrier. Every subsequent stack scanning and *Summary-DS* updates by the mutator will be performed from within the return barrier method. Thus, the mutator scans the stack incrementally.

The collector starts with the ScanRootsByCollector(), which calls InitiateHeader() on the *Summary-DS* as well, and then scans the whole stack, updating the *Summary-DS* with each frame record that was not previously installed by the mutator.

The main synchronization point happens when the mutator or the collector are done with creating a frame record and are trying to install it into the *Summary-DS* structure. Consider first the mutator actions in Algorithm 5. A new stack scanning is started by invoking the InitiateHeader() method, in which a new head is being initialized, implying a new *Summary-DS* structure for this scan. It is assigned with a unique scan number and with the location of the first frame (from which the scan should start). A CAS is used to make sure that only one header is installed for the scan.

Before scanning the next stack frame in the return barrier function (Algorithm 7), the mutator checks if it has already been scanned by the collector. If not, the mutator scans the next stack frame and attempts to add a new FrameRecord to the mutator's tail by applying a CAS operation on the mutatorTail.next pointer. Failing the CAS means that the collector has already installed the frame record and the mutator can just use it. Otherwise, the ScanFrame() method scans the next stack frame and records the list of encountered pointers in the FrameRecord structure, according to the compiler generated maps. Finally, the mutator then updates its mutatorTail and installs new return barrier.

The collector follows a similar procedure, i.e., initiating a header and then attempting to scan each un-scanned frame and add it to *Summary-DS*. The main difference is that it executes this procedure in a loop rather than via a return barrier. If the mutator was faster and the collector needs to skip the already scanned frames, the collector can find the beginning of the last un-scanned frame from the StackPointer recorded in the last FrameRecord. The collector also "applies" the records, i.e., handles the obtained pointers to aid the collection. Normally, this means marking the referenced objects and tracing their descendants.

3.4 An Important Race to Note

The obvious races of initiating the header and installing new frame records have been handled by CASes in the previous description. But there is an additional implicit race to handle. This race occurs when the collector is scanning a frame that the mutator has finished scanning and has started to modify.

In its simplest form this means that the mutator modifies the same stack frame that is currently being scanned by the collector. This results in the collector's reading wrong values from the stack. In a more elaborated form, this race happens when the mutator has popped the frame on which the collector is working and has pushed new different frames. This could result in the collector's both reading wrong values and misinterpreting frame boundaries.

We explain why this race does not create a problem by noting some important invariant and properties. First, the above races only happen when the mutator has started modifying the frame, and that can only happen after the mutator has finished scanning that frame and has already inserted a frame record into the *Summary-DS* structure. It follows that whenever such a race happens, the collector will never be able to insert its frame record into *Summary-DS*. When it attempts to do so, the CAS fails because the mutator has already inserted the record that corresponds to that frame. Second, we maintain a strong invariant that only the mutator modifies the stack itself. The collector can only read values from the stack, but it never modifies it.[3] Thus, whatever the collector does when encountering the above race, cannot affect mutator's execution.

Finally, it remains to make sure that the collector does not get "stuck" because of working on a frame with irrelevant data. To see that, we look closer at the operations that the collector executes during a frame scan. It first uses a (static) compiler map to determine the length of the frame and which of its slots contain pointers. It then copies the values of these slots into its local frame record. This operation terminates after the collector copies all relevant slots. It doesn't matter that the slots values are not valid anymore. These irrelevant values are later ignored when the collector fails to insert this frame record into the *Summary-DS* structure.

3.5 Simple Optimizations

There is a number of possible optimizations that can be easily applied to the above algorithm. First, currently the return barrier lets the mutator move the barrier one frame up at a time. However, if the collector has already scanned some frames, the mutator does not need to scan or install the return barrier for these frames and the barrier can move further up. We let the mutator follow the mutatorTail.next pointer until it is not NULL (in Algorithm 7) and install the return barrier only for the frame immediately following the last frame scanned by the collector.

Another optimization relieves the collector from performing some unnecessary work. The collector does not need to scan the stack frames that were not modified by the mutator since the last collection cycle. To allow this optimization, we must retain the FrameRecords from the previous collection cycle and also maintain a watermark of the last, top-most stack frame modified by the mutator. Such a watermark could be maintained via a similar return barrier mechanism. Notice, however, that for languages that support reference parameters like C# maintaining such a watermark is more complicated and may require a usage of write barrier on the indirect-reference accesses.

3.6 Supporting C#

In languages that support passing parameters by reference (like C# or C++) a reference on the stack can point to a different location on the stack, which in turn points to some heap object. We denote such a reference an *intra-stack* reference. An example is depicted in Figure 3. The location A on the stack, which is a part of stack frame number 3, holds an address of another location B, which is a part of stack frame number 6 further up on the stack, which holds an address of a heap object C. If no special care is taken, the above algorithm could fail to provide a snapshot of the stack of a thread. The reason is that the mutator only scans the most recent frame, but may change older frames using intra-stack pointers. Changing pointers before scanning their frame could foil the snapshot property of the scan, and may violate the collector's safety guarantees.

To solve this problem, we employ a write barrier on indirect stack access. In practice this is obtained by changing the implementation of the special MSIL instruction that executes an indirect access. The barrier is activated by the collector when it raises the ScanRequest flag and can be deactivated for each mutator after the collector has finished scanning its stack. The barrier records the old value of the modified location, thus allowing the recording of a true snapshot of the stack. Notice that the barrier is used only during the stack scanning period and only on indirect stack accesses, i.e., on frames that are not the current one. Thus, the barrier is seldom used in practice, and imposes negligible overhead. In addition, some concurrent garbage collectors already employ such a write barrier during the scanning phase of garbage collection, in

[3] This strong invariant is also maintained with the moving collector described in Section 4

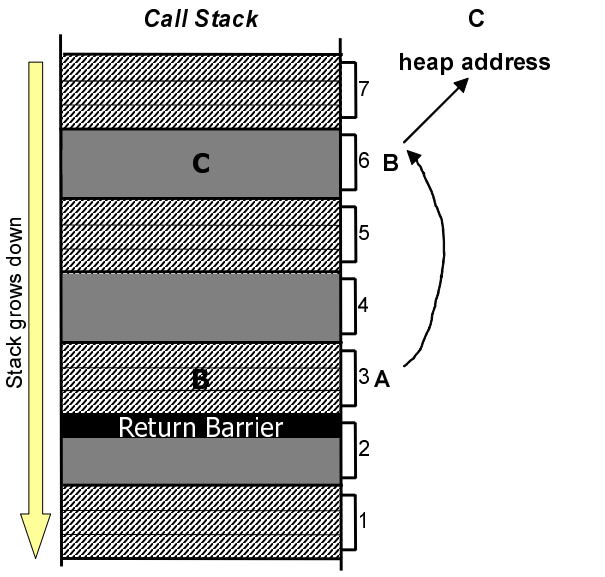

Figure 3. Reference Parameters example

which case the "addition" of the write barrier for stack scanning purposes will not add any overhead.

3.7 Callee-Saved Registers

Callee-saved registers (when they are saved on the stack) create a complication to incremental stack scanning (as well as atomic stack scanning). The reason is that their value is known at the called method, but their type (specifically, whether they are pointers) is known only at the calling routine. Furthermore, if the register has not been used by several recently-executed methods, then its type can only be determined at an unbounded distant frame. The good property that one can use is that saved values of registered are not modified by the saving method or by any method called by it. Thus, these values are not scanned in those later frames. Nevertheless, they should be scanned with the most recent frame that used the registers. The values of callee-saved registers are propagated from stack frame to stack frame. When the scan encounters a frame that modifies a callee saved register, its type can be determined by the compiler maps, and if it is a pointer, its value (propagated from newer frames) is scanned.

3.8 Use for Real-Time Collectors

Real-time collectors must provide a bounded pause time and guaranteed progress pace, so that events can be handled by the application within given deadlines. It is desirable that events can be handled as soon as possible. But predictability is more important than efficiency. Our method allows predicting and arguing about response time. Pause times incurred by the stack scanning are usually short and predictable: they originate from a single stack frame scan, and the length of scanning the worst possible frame can be predicted statically by inspecting the program methods. A source of unpredictability is a possible quick succession of method exits. A method exit incurs an overhead during normal execution, but this overhead may increase due to the return barrier of our method. Again, this increase is statically predictable for any method. Furthermore, exiting a method usually means a completion of an operation, which implies progress. Thus, pace of progress and overhead can be determined by a static analysis of the program code (or by manually inspecting the code).

4. Supporting Moving Collectors

Until now, we have provided a stack scanning mechanism that works well for non-moving collectors. When a collector needs to move objects and update pointers on the stack while scanning it, more care is required. In this section we extend the algorithm to handle moving collectors, i.e., compacting collectors or copying collectors. The non-moving collector has been implemented and measured. But the extension for moving collectors is presented as a design and has not been implemented in our system.

An assumption we make here is that for each pointer in the stack there is a clear action that should be taken when encountering it. For some pointers no action is required, and for others an update of the pointer (and maybe a move of the referent) are required. When moving is required, the information on where to move the object is available from the collector, and similarly, when pointer updates are required, the new value to be written to the pointer is available.

For moving collector, the stack scanning process must make sure that before any pointer is accessed by the mutator, the appropriate operation is applied on it. For example, if an object has moved, then a stack scan may be required to update all pointers on the stack to the object. Indeed, our incremental scan will update all pointers before the mutator accesses any of them.

For reasons that will be specified below, when objects are relocated, we will need to record the new location of the object for later use in the `ApplyRecord()` method or in the next scan. This requires some mechanism for determining the new locations of the objects, typically, a forwarding pointer will do.

4.1 An Overview

We keep the same algorithm and structure as in Section 3, except for the method that scans the stack frames (the `ScanFrame()` method) and the method that uses the pointers that were discovered on the stack frame (the `ApplyRecord()` method).

New problems arise due to the additional relocation of objects. We start by avoiding problematic races using an important design invariant: only the mutator modifies its stack. The collector may read the stack frames and prepare stack frame records with directives to aid the mutator's frame modification, but it cannot perform pointer updates on the stack itself. To allow the frame record to contain such directives, we extend the frame records to hold additional information (as explained below). The invariant that only the mutator modifies its stack resolves many races, that we do not even discuss, but creates a problem not encountered with non-moving collectors. For non-moving collectors, the frame records were only used by the collector for obtaining a list of the roots. The frame records could be discarded after the collector finished. This is not the case now. The frame records are also used by the mutator to update pointers on the stack. Now, suppose the mutator does not use an (old) stack frame, while several collections occur. During these collections, the referent of a stack pointer may be relocated several times, and it should be possible to locate the new copy based on an (extremely) outdated pointer. We thus need to maintain some information throughout the collections to allow such pointer updates. These challenges are handled in what follows.

4.2 The Algorithm

As before, the `FrameRecord` holds the `next` and `StackPointer` fields and a variable length linked list of pointer slots information. For each slot, a triplet is now recorded. Each triplet stores an old pointer value, a new pointer value and the location on the stack (frame index) of this pointer. The recording of the triplets is executed in the `ScanFrame()` method depicted in Algorithm 9. If previous frames records have not yet been applied by the mutator (as discussed above) then a new frame record is created from the previous one, where the old pointers retain their values, but the new

locations get updated. Each frame record has also a pointer to an older frame record, that is kept until the end of the current cycle.

After the `FrameRecord` is created (either by the collector or by the mutator) and before the mutator accesses this stack frame, it updates the addresses in this frame with the new to-space addresses recorded in the corresponding `FrameRecord`, as described in the `ApplyRecord()` method in Algorithm 9. Note that since the mutator needs to update every stack frame, the optimization of the mutator skipping a number of return barriers suggested in section 3.5 for marking is not possible for moving collectors.

Algorithm 9 Moving Collector with Lazy updates

```
FrameRecord ScanFrame(int* addr)
  FrameRecord prev = Summary-DS.getPrevCycleRecord(addr);
  if prev.wasApplied() then
      scan the stack, follow forwarding pointers if necessary,
      and record addresses and their stack-location in curr;
  else
      scan the addresses in prev, follow forw. pointers if necessary,
      and record addresses and their stack-location in curr;

ApplyRecord(FrameRecord record, bool isCollector)
  if isCollector then
      mark and trace addresses in the record;
  else  // Mutator execution
      if !record.wasApplied() then
          update the stack with addresses in the record;
          record.setApplied();
```

The relevant pointers of the stack frame are deduced from the `FrameRecord`s of the previous cycle *Summary-DS*. The details are depicted in the updated version of methods `ScanFrame()` and `ApplyRecord()` in Algorithm 9. Both the collector and the mutator use a previous cycle *Summary-DS* in conjunction with the heap while scanning the stack as part of the current scanning cycle. The action depends on whether the stack frame has already been updated or not. Therefore, after the mutator updates every frame, it marks the corresponding `FrameRecord` as applied. During the scan, if the corresponding `FrameRecord` from the previous cycle was already applied (the stack was updated), the addresses can be deduced from the stack. Otherwise, they must be deduced from the previous cycle `FrameRecord`. If pointer updates are required in this cycle, the new to-space address is also deduced by following the forwarding pointer.

4.3 Alternative Solutions

The latter complications due to a lazy stack updating could be resolved differently, for example by making the mutator finish the updates of the stack by the end of each collection as in [5]. This would require a work-based collector scheduling, which may in turn lead to low mutator utilization.

Another possible solution is for the collector to help the mutator fix the frames. However, such a solution would potentially create numerous races and require a more complicated and costly synchronization and versioning schemes (possibly similar to the solution proposed in [22] for heap).

4.4 Supporting C# for Moving Collectors

Further complications arise when dealing with moving collectors in the presence of C#-style intra-stack references. We propose a solution for this case that works only with collectors that follow an eager-update policy. Such collectors update each stale pointer they encounter before writing it to the stack. Thus, it is guaranteed that all stack pointers are updated. This allows such systems to use stack pointers without any read or write barriers.

Like in Section 3.6, we will need to employ a special memory barrier on all indirect stack-accesses. When an intra-stack pointer is used to indirectly reference a heap object, we will check the referenced object for a possible forwarding pointer and if it exists, we will follow it.[4] We will specify what the mutator does when it updates a frame via the return barrier, and we will then argue that the simple intra-stack read-barrier described above suffices.

When the mutator updates a frame (using a previously-prepared frame record), it may encounter an intra-stack reference. For each such encountered intra-stack pointer, the mutator also updates the stack frame of the target address. Namely, when an intra-stack pointer references a stack address that belongs to an older frame F, the mutator applies the currently available frame record for F (in addition to updating the current stack frame). Normally, the currently available frame record for F is an updated one, and the resulting frame F is properly updated and can be used (even without the forwarding pointer barrier proposed above) for intra-stack access. However, when the update of the older stack is executed during the stack scan, there is a chance that the mutator will use a frame record that was prepared during the previous collection cycle. This can happen if the current stack frame is being applied, but the older stack frame F has not yet been scanned by the collector and its new frame record has not yet been prepared. To deal with this case, we must employ the forwarding read barrier on indirect intra-stack accesses. Furthermore, we add an additional task for the mutator. After the collector finishes the stack scan of the entire mutator stack, it make the mutator re-apply the frame records. This is done by restarting the return barrier execution, placing it at the most recent frame. At this point, all frame records exist, and so the mutator will never have to prepare one. But it will have to apply all frames according to their updated version on the corresponding frames. The implication is that, the mutator will have to repeat its work for some of the frames. At worst case, the mutator will apply the records to all frames twice, but the gain is that at the point of time in which it starts the second pass, we know that it will never access a stale pointer on the stack. This means that at that point we can start reclaiming objects.

We now argue that using forwarding for indirect heap access via intra-stack pointers is possible and correct during the stack scan until the point in which the collector finishes the scan and directs the mutator to re-apply the frame records. During this time, the mutator may use a stale pointer in an older frame and will need to use forwarding. Such forwarding requires that the old version of the object has not been reclaimed (or reallocated). Otherwise, the forwarding pointer in that object cannot be used reliably to reach the new location of the same object. Note first, that the outdated pointer is not too old. It was computed from a frame record of the previous collection. This means that it references the location of the object that was correct at the end of the previous collection and at the start of the current one. Nevertheless, the referent may have moved in the current collection cycle. To ensure that a reference to an object from the previous cycle can be still used for forwarding during the stack scan, we require that no object is reclaimed (and re-allocated) while the stack scan is in progress. Once the collector finishes the scan and sets the return barrier for the mutator again, we can be sure that further use of the stack by the mutator only accesses updated pointers and at that point old copies of the objects can be reclaimed.

In this conference version of the paper we do not discuss specific BARTOK additional complications. Particularly, some more

[4] Typically, collectors maintain a self pointer at the forwarding pointer location when forwarding is not required. This allows following the pointer without checking whether a forwarding pointer is relevant. This kind of barrier is called a *Brooks* barrier.

work is required to find the frame corresponding to a given address and to find a frame record of a given frame. To solve these problems in the BARTOK setting, we make more use of the "double pass" solution above to prepare information at the initial collector pass and use it at the second mutator pass. Details are omitted. An additional issue might arise in languages that allow pointers to the middle of heap objects (like C#). For real-time moving collectors, for which the scan time must be bounded, an additional small table may be added to guarantee that objects holding these pointers can be identified in a bounded time.

5. Implementation and Measurements

The incremental stack scanning mechanism has been implemented in BARTOK runtime.

Below we report on our experiments performed using a configuration where the scan increments are single stack frames, and where each `FrameRecord` data structure is dynamically allocated from the heap. The use of the smallest possible scan increments will demonstrate that extremely short pause times can be achieved. The use of dynamic allocation of the `FrameRecord` data structures is a weakness of our current implementation and will result in pause times that are significantly larger than necessary.

The stack scanning mechanism has been used with a non-copying concurrent mark-sweep collector in the style of the DLG collector [15, 14].

All experiments have been performed on a Dell Precision 490 workstation with two Intel Core 2 Duo 5150 processors (for a total of 4 cores) @ 2.66GHz and 4GB of memory running a 64-bit version of Windows Vista Enterprise. The programs are run at normal priority. The machine was run in a normal environment, including being attached to a network, running anti-virus software, etc. All applications were compiled to x86 machine code.

We report on the behavior of the stack scanning mechanism for two large programs and a small artificial test program. The first large program is the BARTOK compiler compiling itself. Due to substantial inlining of methods, the stack frames are very large. The stack depth is typically 10-30 frames, but will at times reach into the hundreds. The second large program is a variant of JBB ported to C#. The small artificial program consists of a deeply recursive function that triggers a garbage collection in the leaf call. The program exhibits very small stack frames, but will put substantial pressure on the incremental stack scanning mechanism by unwinding the entire call stack immediately after triggering the garbage collection.

For the small artificial program, the pause times experienced by the mutator thread are shown in the first two columns of Table 1. The majority of the pause are on the order of 1 microsecond in duration. The results show that extremely short pause times are indeed possible.

We measured pause times for the JBB program in a configuration where it used 3 warehouses. When simulating 3 warehouses, there are 3 very active threads in the application. On a 4 core machine, this leaves one processor available to do other work, which in this case amounts to performing garbage collection, including the stack scanning. The results are shown in the last column of Table 1. The results indicate that even for realistic programs, the pause times are very short.

Table 2 shows for each thread the number of collections, the number of stack frames scanned by the mutator itself, and the number of stack frames scanning by the garbage collector.

Prior to using the incremental stack scanning implementation, we would once in a while observe mutator pause times in the order of 10 milliseconds. We have not seen such big pause times for mutators when using the incremental stack scanning mechanism.

Duration	Incremental			Atomic		
	small	JBB	BARTOK	small	JBB	BARTOK
$1\mu s$	6909	3037	259	0	0	0
$2\mu s$	75	1596	138	0	0	0
$3\mu s$	3	424	60	0	0	0
$4\mu s$	0	571	40	0	0	0
$5\mu s$	0	283	6	0	0	0
$6\mu s$	0	137	11	0	0	0
$7\mu s$	0	66	12	0	0	0
$8\mu s$	2	130	27	0	0	0
$9\mu s$	0	218	31	0	1	0
$10\mu s$	0	176	50	0	16	0
$11\mu s$	0	55	21	0	35	0
$12\mu s$	0	27	18	0	45	0
$13\mu s$	0	23	7	0	25	0
$14\mu s$	0	4	2	0	24	0
$15\mu s$	0	2	5	0	15	0
$16\mu s$	0	2	0	0	14	0
$17\mu s$	0	1	1	0	12	0
$18\mu s$	0	1	1	0	23	0
$19\mu s$	0	1	2	0	46	0
20–$29\mu s$	2	2	6	0	201	1
30–$39\mu s$	1	3	7	0	5	17
40–$49\mu s$	0	1	13	0	0	14
50–$59\mu s$	0	0	14	0	0	3
60–$69\mu s$	0	1	5	0	2	16
70–$79\mu s$	0	0	3	0	0	15
80–$89\mu s$	0	0	1	0	1	4
90–$99\mu s$	0	0	2	0	0	1
100–$200\mu s$	0	0	0	0	2	5
200–$300\mu s$	0	0	0	0	1	0
300–$999\mu s$	0	0	0	1	4	0

Table 1. Pause times for the small artificial program, for JBB with 3 warehouses, and for the BARTOK compiler. Number of occurrences for each pause time.

Thread	Collections	collector frames	mutator frames
BARTOK	85	1475	180
JBB 1	415	3275	103
JBB 2	414	3359	34
JBB 3	414	3330	177

Table 2. Distribution of work for the BARTOK compiler and for the three JBB threads.

6. Related Work

Many on-the-fly collectors are described in the literature [26, 27, 12, 15, 14, 16, 17, 20, 3]. All of them stop one thread at a time to scan its stack. Each stack scan must be atomic to guarantee correctness of the collection. On the fly *moving* collectors such as [19, 22, 21] require that the pointers in the thread stack are updated atomically to reflect the move of objects from their old location to the new one. Our stack scanning algorithm can be used with any of these algorithms to incrementally scan the stack, while appearing to the garbage collector as an authentic atomic snapshot.

Incremental stack scanning for copying collectors was first mentioned in [5]. It uses read barrier on stack accesses to guarantee that the latest copy of an object is accessed. To guarantee that stack tracing will be finished before the next spaces flip, a predefined number of locations, based on statistics gathered from previous flip, is

traced and forwarded during each allocation. However, the stack is not scanned concurrently by the mutator and the collector, lock-freedom is not discussed, and no support is provided for modern stacks with unmanaged code.

The reason that the stack must be scanned atomically for mark-sweep collectors whereas the heap can be scanned incrementally, is that practical collectors do not use memory barriers for stack modifications. Stack modifications are frequent and memory barriers are considered highly costly for them. Nevertheless, one can eliminate the requirement for atomic stack scanning, by simulating the stack in the heap. This solution is exercised in the Jamaica VM [25, 24]. There, at most one Java thread is executing at any time while all other Java threads are stopped at synchronization points. The Jamaica VM does not support lock-freedom due to the usage of synchronization points and its design does not support scalability on multiprocessors. A similar approach is exercised by Appel et al. [1] who copy the stack to the heap before the collection start and then use normal barriers on it.

The usage of stacklets is proposed in [9]. The stack is divided into fixed size stacklets (being one or more stack frames), allowing the collector to scan and replicate all but the currently used stacklet. The cooperation between mutators and the collector thread is not lock-free. The mutator is blocked until the collector finishes copying the latest stacklet.

The return barrier mechanism was first used in [18] in the context of debugging optimized code, to allow lazy dynamic deoptimization of the stack. Cheng et al. [10] used some version of a return barrier in order to maintain a watermark on the stack usage. Functional languages use deep call stacks, that tend not to be accessed between different collection cycles. Thus, the parts of the stack that did not change from the previous stack scan do not need to be rescanned. A return barrier is used to maintain the lowest frame that was modified.

The return barrier mechanism was also previously used for the LISP collector in [28]. In this work the stack is scanned incrementally, but not concurrently (i.e., by the mutator thread only). The mutator scans a frame once in a while and the collector must wait until it finishes. The return barrier is used as a precautionary measure – when the mutator reaches some stack region that was not previously scanned, the execution is trapped and the frame is scanned. This work does not support concurrent scan (and thus does not need to deal with synchronization), and in particular, it cannot collect the heap if one of the mutator threads blocks on a system call.

7. Conclusion

We have introduced a stack scanning mechanism that is concurrent, incremental, and lock-free. Our mechanism can deal with modern multithreaded programs that perform system calls to unmanaged code, and with C#'s stack intra-pointers. This algorithm is targeted at runtimes that support highly responsive application. We have implemented this algorithm in the BARTOK runtime and compiler for C#, and the results show that the pause times of on-the-fly collectors for standard benchmarks can decrease substantially.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-Time Concurrent Collection on Stock Multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[2] H. Azatchi and E.Petrank. Integrating Generations with Advanced Reference Counting Garbage Collectors. In *CC*, pages 185–199, 2003.

[3] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An On-The-Fly Mark and Sweep Garbage Collector Based on Sliding View. In *OOPSLA*, 2003.

[4] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *PLDI*, 2001.

[5] H. G. Baker. List Processing in Real-Time on a Serial Computer. *CACM*, 21(4):280–94, 1978.

[6] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A Parallel, Incremental, Mostly Concurrent Garbage Collector for Servers. *ACM TOPLAS*, 27(6):1097–1146, November 2005.

[7] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. *ACM SIGPLAN Notices*, 38(11):344–358, 2003.

[8] H. J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[9] P. Cheng and G. Blelloch. A Parallel, Real-Time Garbage Collector. In *PLDI*, pages 125–136, 2001.

[10] P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-Driven Pretenuring. In *PLDI*, pages 162–173, 1998.

[11] L. Peter Deutsch and Daniel G. Bobrow. An Efficient Incremental Automatic Garbage Collector. *CACM*, 19(9):522–526, 1976.

[12] 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. *Lecture Notes in Computer Science, No. 46*, 1976.

[13] 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. *CACM*, 21(11):965–975, November 1978.

[14] D. Doligez and G. Gonthier. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *POPL*, pages 70–83, 1994.

[15] D. Doligez and X. Leroy. A Concurrent Generational Garbage Collector for a Multi-Threaded Implementation of ML. In *POPL*, pages 113–123, 1993.

[16] T. Domani, E. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, E. Petrank, I. Yanover, and Y. Levanoni. Implementing an On-the-fly Garbage Collector for Java. In *ISMM*, pages 155–166, 2000.

[17] T. Domani, E. Kolodner, and E. Petrank. A Generational On-the-fly Garbage Collector for Java. In *PLDI*, pages 274–284, 2000.

[18] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *PLDI*, 27(7):32–43, 1992.

[19] R. L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE Conference*, June 2001.

[20] 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.

[21] F. Pizlo E. Petrank and B. Steensgaard. A Study of Concurrent Real-Time Garbage Collectors. In *PLDI*, June 2008.

[22] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: A real-time garbage collector for modern platforms. In *ISMM*, pages 159–172, October 2007.

[23] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. In *ISMM*, pages 143–154, 2000.

[24] F. Siebert. Hard Real-Time Garbage Collection in the Jamaica Virtual Machine. In *RTCSA*, page 96. IEEE Computer Society, 1999.

[25] F. Siebert. Constant-Time Root Scanning for Deterministic Garbage Collection. In *International Conference on Compiler Construction (CC)*, pages 304–318, April 2001.

[26] Guy L. Steele. Multiprocessing Compactifying Garbage Collection. *CACM*, 18(9):495–508, September 1975.

[27] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *CACM*, 19(6):354, June 1976.

[28] T. Yuasa, Y. Nakagawa, T. Komiya, and M. Yasugi. Return-Barrier. In *Proc. of the International Lisp Conference*, 2002.