Reducing Pause Time of Conservative Collectors

Toshio Endo
National Institute of Informatics
2-1-2 Hitotsubashi Chiyoda-ku
Tokyo 101-8430, Japan
endo@nii.ac.jp

Kenjiro Taura
Graduate School of Information Science and Technology
University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo 113-0033, Japan
tau@logos.t.u-tokyo.ac.jp

ABSTRACT

This paper describes an incremental conservative garbage collector that significantly reduces pause time of an existing collector by Boehm et al. Like their collector, it is a true conservative collector that does not require compiler cooperation but uses virtual memory primitives (page protection) of operating systems for write barriers. While much successful work has been done on incremental collectors in general, achieving small pause time of the order of a few milliseconds in such uncooperative settings has been challenging. Our collector combines several ideas that bound pause times without introducing significant overheads. They include: (1) bounding the number of dirty (writable) pages during concurrent marking, (2) adaptive repetition of concurrent marking phases, and (3) allocating objects in black (marked) in later stages of a collection. With these techniques, we have achieved the maximum pause time of 2.6–4.5ms for five application benchmarks on 400MHz UltraSPARC processors. They are up to forty times shorter than the basic incremental collector similar to Boehm et al.'s. The overhead (total work time) of our collector is 1.2–53% to the stop-the-world collector and 9% or less to the basic incremental collector.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – memory management (garbage collection)

General Terms

Performance

Keywords

conservative garbage collection, concurrent garbage collection, parallel garbage collection, memory management

ISMM'02 June 20-21, 2002, Berlin, Germany.

1. INTRODUCTION

Conservative garbage collector, first invented by Boehm et al. [8], has been very successful. It can automatically reclaim memory without involved cooperation from the compiler, such as using a particular representation of objects or generating "pointer maps." Both theoretical analyses and empirical evidences show it is very efficient [5, 7, 10, 14, 15] both in terms of space and time. It has been used in many programming language implementations and C/C++ applications. Users (application or compiler writers) can easily "built-in" a garbage collection without imposing almost any condition, complexity, or performance penalty to the rest of the system.

There are at least two rational reasons why prospective users might still prefer implementing custom collectors to using already robust and efficient conservative collector library available today [4, 21]. One is that allocation is more expensive than the linear allocation used by true copying collectors. The other is that conservative garbage collectors are generally harder to make realtime (or incremental). By realtime or incremental, we mean a single pause time of the application introduced by the garbage collector is sufficiently small. A common threshold is a single-digit milliseconds, making many multimedia applications work non-disruptively. We do not know any existing conservative collector that satisfies this with today's contemporary CPUs (we used UltraSPARC 400MHz processors for our experiments). This paper is about solving this problem and making conservative collectors more widely useful.

Of course, realtime or incremental collections [2, 11, 27] have been extensively studied in the past, and the basic ideas to reduce pause time is known. See [19, 26] for good surveys. The basic idea is to mark objects incrementally, maintaining some invariants with appropriate read/write barriers so as not to lose any live object. Most previous work, however, assumes the compiler is able to insert fine-grain read/write barriers as necessary, making them less amenable to our problem setting.

For conservative collectors, necessary barriers are induced by page protection hardware and its operating system support. Theoretically speaking, there is not qualitative difference between barriers induced by compiler-inserted code and those by page protection signals, but since the cost of protecting a page is high and the protection can be introduced only with a single page granularity or larger, design choices

are restricted in many ways. Achieving both efficiency (low overhead) and small pause time has thus been challenging.

We are aware of three collectors [2, 6, 23] that focus on incremental collection with page-wise protection only. The algorithm by Appel, Ellis, and Li [2] uses read barriers, so it will incur much overhead in practice. Furuso et al.'s collector [23] uses only write barriers. However, because it is based on snapshot-at-the-beginning algorithms, it may incur significant space overhead. "Mostly parallel GC" by Boehm et al. [6], which our design is based on, is more space-efficient, but it does not have a good-enough pause time both in theory and in practice, probably in favor of making overhead (total work) small. In our experiences with this collector, pause times of the order of 100ms (on UltraSPARC 400MHz) are not unusual. We analyze the problem in depth and discuss why it is not a trivial engineering issue to achieve both efficiency and small pause times. Our collector significantly improves pause times without introducing much overhead. In our five application benchmarks, the maximum single pause time is 4.5ms on the same processor.

Our collector implementation is based on our previous work on scalable parallel collectors [14, 15]. The collector presented herein is a "concurrent and parallel" collector [9, 13]. It is "concurrent" in the sense that collector threads run concurrently with the application threads. It is "parallel" in the sense that multiple collector threads participate in a single collection. Our experimental results indicate that operating systems used for our experiments have a bottleneck in virtual memory operations for parallel applications. As a result, our collector sometimes pauses individual threads, rather than all threads, for a long time (up to 15ms). We do not have a solution for this other than changing operating system implementation.

The rest of the paper is organized as follows. Section 2 reviews principles behind reducing pause times of collectors and discusses why it is challenging to reduce them in conservative settings. Section 3 describes Boehm et al.'s collector our design is based on. Section 4 describes our collector; it explains our multiprocessor support and then details techniques to reduce pause times. Section 5 evaluates the performance of our collector, by using both sequential and parallel programs. Section 6 mentions related work, and we conclude in Section 7.

2. PRINCIPLES BEHIND BOUNDING PAUSE TIME

2.1 Basics

This section reviews basic principles in reducing or bounding pause time of tracing collectors and discusses why even incremental collectors sometimes experience a long pause in practice. Traditional stop-the-world collectors stop the user program, mark all objects reachable from the root, and collect unreachable objects. They stop the user program to mark all objects atomically (i.e., without seeing changes in the object graph). Incremental collectors shorten pauses by making the collector mark the object graph while it is being changed by the user program. Such algorithms must ensure they never conclude objects that are actually reachable are not.

There are two major approaches to ensuring the correctness of incremental collectors, namely, "snapshot-at-the-beginning" and "incremental update." The former, including the best-known algorithm by Yuasa [27], achieves the correctness by ensuring that the collector visits all objects reachable in the snapshot at the beginning of a garbage collection and all objects created thereafter. To implement this, it suffices to trap pointer updates and *captures all pointers that are being overwritten*, and mark all objects created during marking. When marking is finished, the user program can be resumed immediately. Therefore this approach does not have much difficulty in reducing pause times. The problem of snapshot-at-the-beginning is not in pause times, but in the associated overhead, especially when implemented with virtual memory approach. With this approach, a write fault handler must copy the entire page at which the write fault occurs. If the application writes to many pages, time and space overhead for this copying may be prohibitive. Therefore, this paper does not discuss this approach any further.

The other approach, incremental update, is introduced by Dijkstra et al. and Steel [11, 17]. To describe this approach, we introduce the well-known tri-color abstraction. During a collection, objects are one of three colors, white, grey, or black. White objects have not been visited by the collector. Grey objects have been visited, but their direct children may not be. Finally, black objects have been visited and their direct children are grey or black. For the sake of our discussion, the root can be regarded as an object colored grey at the beginning of a collection.

The basic invariant of a tracing collector is there are no black to white pointers at any instant. The typical marking process can be viewed as a process of picking up a grey object O, coloring its white children grey, and then blackening O, thereby maintaining the invariant. When we reach the state in which the root is black, there are no grey objects in the world, and the tri-color invariant (no black $\rightarrow$ white) holds, we know all objects reachable from the root at that moment are black, so all white objects are dead (unreachable).

An incremental update algorithm maintains this invariant by trapping pointer writes to objects. When it traps a pointer write that is about to create a pointer $a \rightarrow b$, it either (a) greys a if b is white, or (b) greys b if a is black. Either approach is fine, but observe that (a) may starve the collector because the application might repeatedly write to a, keeping a grey forever. Also observe that with virtual memory approach, (a) is the only practical choice, because if we take approach (b), object a remains black after the update, so further updates of pointers within a must still be trapped hereafter. This means that the page that holds object a must remain protected. Updating object a while maintaining the page protected is already difficult, and even if this is possible, it incurs high overheads if a is updated often. With approach (a), on the other hand, object a becomes grey, so no further writes to a need be tracked so long as a stays grey.

2.2 Root Set and Incremental Updates

Strictly maintaining the above invariant is too costly, however, especially for frequently written objects. One such extreme is a part of the root set such as registers and the current stack frame. Recall that a root is regarded as an object and the correctness of the garbage collector relies on the fact that the root is black when finishing a collection.

Maintaining the invariant for the root means we must actually trap every reads from memory (*read barrier*), because reads are actually writes to registers. Since this is prohibitively expensive in practice, practical implementations use an alternative. They only trap *writes to heap objects*, and assume the portion of the root writes to which are not trapped may revert to grey at any time (unless the application is stopped). We call such portion of the root *non-trapped* area. Typically, it is registers and stacks.

In this framework, when a collector finds there are no grey objects (other than the non-trapped area), it stops the user program, examines the non-trapped area again as if it is a grey object, and checks if all direct children are black. If this is the case, a collection cycle is finished. There are several possible actions to take in case the check fails. It may recursively mark all descendents, or resume the user program again concurrently with marking. We call this final pieces of work "the final marking".

Time for a final marking is not obviously bounded, whether we can use a cooperative compiler or not. A scenario that often occurs in practice is that the application creates a big tree (graph) of objects, and makes it reachable only from a register after a tracing begins. Since writes to registers are usually not trapped, the collector finds these objects only in a final marking. Note that snapshot-at-the-beginning approach would find it during normal tracing, because newly allocated objects are immediately regarded as "marked".

The situation is further complicated if we do not rely on compilers that emit write barriers for individual writes, but instead rely only on page protection. The main difference is that, using compilers, it is efficient to trap *all* writes, including those to the same object. Because of this, the usual strategy is that, when an object reverts to grey by a pointer update, the collector simply continues working and makes it black again sometime later in the same marking phase. Note that if another pointer to a white object is written to it again, that write is trapped too without extra cost. With only page protections, on the other hand, once an object reverts to grey, it is very expensive to make it black again, because it incurs the cost of visiting all marked objects in the page and write-protecting it. We also have the above starvation problem in a worse way; a single write to a page by the application invalidates all the collector's work of blackening the page, namely, visiting all marked objects in the page and coloring all direct children grey.

Hence, the usual strategy with page protection is that, once an object reverts to grey, the collector leaves it (and all objects in the same page) grey until the final marking. Subsequent writes to them are not trapped. Such pages, called *dirty pages*, are examined in the final marking, in the same way registers and stacks are. In essence, *incremental collectors with page protection treat dirty pages as if they are in the non-trapped portion of the root.* This means a final marking without cooperative compilers sees a much bigger "pseudo root set" than with cooperative compilers. With cooperative compilers, we can easily limit the size of the non-trapped area. For example, it is trivially bounded if the only non-protected area is registers. Also practically important, even if we take no particular efforts to make it bounded, it is usually not very big anyways (they are typically registers and stacks). With only page protections as a means to write barriers, on the other hand, the problem is severer because the number of non-protected pages easily grows very large, which is proportional to the number of actively written pages. This is a part of the reason why it has been a challenge to implement a good incremental conservative collector.

In the following sections, we first examine Boehm et al.'s conservative collector, which our work is based on, and describe how we modified it to reduce pause times without introducing much overhead.

3. MOSTLY PARALLEL GC

Boehm et al. [4, 6] have implemented a concurrent mark-sweep collector that does not require cooperative compilers (i.e., compilers that emit write barrier code for collectors). Their collector is based on an incremental update algorithm; it needs to rescue white objects that may be reachable, at the end of a collection cycle. This section describes features of their collector relevant to this paper and discusses its pause times.

3.1 Heap Structure

The user program consists of one or several threads that share a single heap. To fulfill allocation requests from threads, the system maintains *free lists* to hold unused regions. Threads can access any allocated object in the heap. Each object has a corresponding flag called *mark bit*. Mark bits are placed in bitmaps out of the heap. We call objects whose mark bits are set "marked" and other objects "unmarked." The collector maintains an array named *mark stack* to keep track of marking task described below. The mark stack contains references to wave-front objects of recursive tracing. Unlike snapshot-at-the-beginning algorithms, new objects allocated during GC cycles are regarded as "unmarked." Therefore, unlike snapshot-at-the-beginning approach, objects born during a collection and dead at the end of the collection may be collected immediately.

3.2 Collection Algorithm

Any incremental mark-sweep collector requires notification when a user thread updates a heap object. This interaction is called a *write barrier* and is introduced to ensure that the collector finds all reachable objects. Mostly parallel GC implements write barriers by making all heap pages read-only with `mprotect` system call at the beginning of a collection. When a user thread updates an object in a protected page, the operating system invokes a signal handler defined by the collector. The collector maintains a data structure named *dirty pages set* to record updated pages.

A GC cycle consists of following actions:

Initial and protect phase: When a GC cycle is started by thread T, mark bits of all objects are cleared. Then T sends signals to all other user threads to suspend them. Threads that received a signal inform the collector of their local roots (stack pointer and contents of registers). T pushes all objects directly pointed to by global variables and roots of all threads onto the mark stack. It makes all pages in the heap read-only by the `mprotect` system call and clears the dirty pages set. Then all threads are resumed.

Concurrent marking phase: Each user thread performs a fixed amount of marking task whenever it allocates a new object. Marking objects can be done without

synchronization with other threads not performing allocation at the same time. The thread pops a reference to an object from the mark stack, pushes its unmarked children, and sets their mark bits. When it performs a certain amount of marking, it returns to the user program.

When a thread finds the mark stack empty, the concurrent marking phase is finished; it goes on to the final marking phase.

Write fault handler: During the concurrent marking phase, the user program may update objects in protected pages. Then the operating system invokes a signal handler. The handler obtains the address of the updated page and adds it to the dirty pages set. Then it makes the page writable by `mprotect` and resumes the user program.

Final marking phase: When the collector finds the mark stack empty (i.e., no grey objects), it suspends all threads. Then it performs marking recursively from roots and marked objects in dirty pages. As described below, this phase may take a long time.

Concurrent sweeping phase: Sweeping is performed concurrently with application threads as they allocate new objects. The collector inspects mark bits of all objects and pushes unmarked objects onto free lists.

This collection algorithm sometimes exhibits a long pause (bounded only by a proportion to the heap size), due to the following problems.

Heap protection in the initial phase: It suspends all the threads while `mprotect`ing the entire heap. While protecting the entire heap is not as expensive as protecting all pages individually, it still asymptotically takes time proportional to the number of pages protected. This problem is important especially for parallel programs on multiprocessors, because the cost of protecting memory region shared by many processors is large as shown in Table 1.

This can be easily fixed by `mprotect`ing the heap concurrently with the application. Observe that this does not cause any correctness issue as long as we protect each page before any object in it is marked black.

Recursive marking in the final marking: Before the collector starts sweeping, it must ensure that all reachable objects have been marked. This may not be the case immediately after the concurrent marking phase, because user programs may have hidden some references to white objects. As we have described, only non-trapped area (the roots and dirty pages) may have such references. Thus the final marking phase scans the area to find references to white objects, and marks them and their descendants. When no unmarked objects are found fortunately, the cost of the final marking phase is proportional to the size of the non-trapped area, which is $O(R + D)$, where R denotes the size of the roots and D the number of dirty pages. The cost in the worst case, however, may be much larger. Suppose that the collector finds a single unmarked object in scanning, and it is the root of a big tree whose size nearly dominates the entire heap. In this case, the cost can be $O(R + M)$, where M denotes the heap size.

We have described the early Mostly Parallel GC algorithm presented in [6], in which the final marking always recursively marks all reachable objects, no matter how long it takes. More recent GC library that Boehm et al. have released [4] has an improved strategy to shorten pauses; it aborts the final marking and resumes the concurrent marking phase if the final marking takes more than 50ms. We will describe a similar technique in Section *4.2.2*. However, unlike ours, their implementation still limits the number of retries to two, so the second final marking may take a long time.

Region size	1CPU	4CPUs	12CPUs
16MB	6.34(ms)	10.0(ms)	17.1(ms)
32MB	12.4	20.1	34.3
64MB	24.8	41.0	69.8

Table 1: The execution time of `mprotect` system call on Sun Enterprise 4500 (Ultra SPARC processors 400MHz × 14, Solaris 8). Region size (16, 32, or 64MB) is the size of the region to protect and CPUs the number of CPUs actively using these pages. The cost of virtual memory operations increase as the number of CPUs actively using the affected pages.

This collector algorithm exhibits another performance problem, when it is used by multithreaded parallel programs on multiprocessors. Because only a single thread can perform collection tasks, the collector may not be able to catch up the allocation speed of user programs if we have a large number of threads. This problem not only degrades the throughput of applications, but may effectively prolong the pause times because threads are blocked until collection tasks are advanced enough. We can reduce the bottleneck of the collection by using *multiple collector threads*.

4. OUR GC ALGORITHM

This section describes our extended collector algorithm that solves the problems of the basic algorithm described above. First, our collector supports multiple collectors in order to achieve high scalability on multiprocessors. As shown in Figure 1, several threads may simultaneously perform collection tasks in parallel. All user threads are suspended during the initial phase and *termination check phase* (which is a substitute for the final marking phase). In other phases, collector runs concurrently. Our parallel method is briefly described in Section 4.1. Secondly, we shorten the initial phase by protecting the heap concurrently in a separate phase just after the initial phase. Last but not least, we reduce the pause times induced by the final marking phase. In Section 4.2, we describe techniques to shorten the final marking.

4.1 Support for Multiprocessors

Our collector is a concurrent parallel collector like Cheng et al.'s[9]; it supports multiple collector threads to efficiently exploit multiprocessors. As Halstead [18] has described, tasks of tracing collectors involve parallelism. Our collector exploits the parallelism as follows.

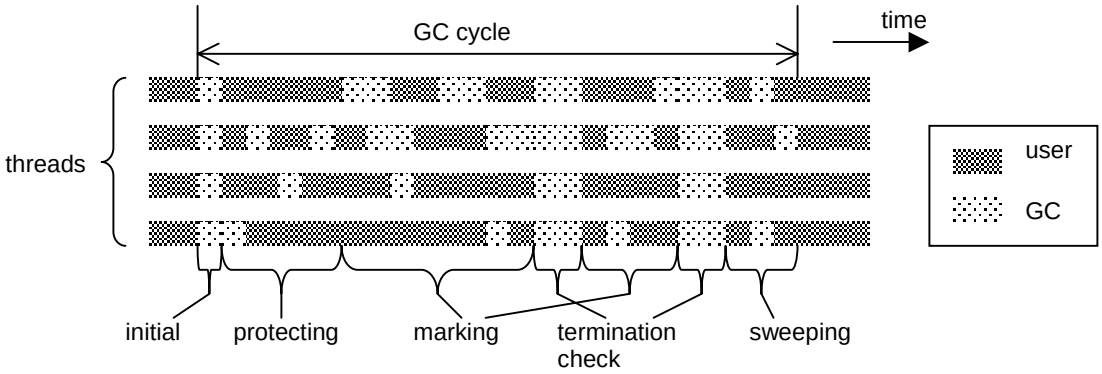

Figure 1: Overview of a collection cycle in our collector. Collection tasks are done concurrently with applications except in the initial phase and the termination check phase, during which all user threads are suspended. Several threads can perform collection tasks in parallel except in the protecting phase. The concurrent marking phase and termination check may repeat several times.

Parallel marking: In many programs, the object graph in the heap has enough width, thus marking the object graph in parallel has an ample amount of parallelism. Each thread in our collector maintains its own mark stack and traces objects from its roots (i.e., stack and registers). A race condition arises when multiple threads visit a single object simultaneously. We resolve it by updating the mark bit of an object with an atomic compare-swap instruction.

We have described elsewhere [14, 15] that a dynamic load balancing is the key to achieving scalability. The dynamic load balancing is performed by exchanging contents of mark stacks among threads.

Each thread also maintains its own dirty pages set to reduce bottleneck in handling dirty pages.

Parallel sweeping: While checking mark bits of distinct objects can be done independently, a contention occurs when multiple collector threads try to add unmarked objects into a single free list. We can alleviate this bottleneck by having each thread maintain its own free list.

Unfortunately, we found that the concurrent protecting phase cannot be done in parallel, at least in our environments. This is because Solaris operating system, which we used for our experiments, seems to serialize `mprotect` system calls invoked by multiple threads, even if each thread operates on distinct memory regions. We have observed this problem also in Linux and IRIX. For the same reason, write fault handlers are also serialized.

4.2 Reducing Pause Times of the Final Marking

In the basic algorithm described in Section 3, the final marking phase may prolong pause times. Tasks in the final phase are classified into two groups: (1) scanning non-trapped area, and (2) marking all unmarked descendants. We reduce task (1) by scanning a part of dirty pages eagerly, and postpone task (2) by repeating concurrent marking and termination check.

The quick termination check is similar to the final marking. It visits objects pointed to by non-trapped area (the root and dirty pages), recursively. Unlike the final marking, however, it aborts after a prescribed amount of work has been performed and then resumes the user program and the concurrent marking. We repeat a concurrent marking followed by termination check until the check finishes quickly. As we have described, Boehm et al. have already introduced the idea to repeat concurrent marking up to twice. Our algorithm can be seen as a more sophisticated extension to this strategy. We do not have a fixed bound on the number of retries, but have a strategy so that a quick termination check is likely to succeed in a small number of retries.

4.2.1 Bounding Dirty Pages

In the basic algorithm described in Section 3, we leave all pages updated during a collection dirty until the final phase. The overhead of this strategy is low because it invokes at most two virtual memory operations (one protection at the beginning of a collection and one unprotection on the first write) for each page in a collection cycle. It is too expensive to clean dirty pages immediately after mutation, because it incurs more page protection costs. However, the larger the number of dirty pages is, the longer the pause time in the final marking tends to be.

We take a compromise; we *bound the number of dirty pages*. We inspect the size of the dirty pages set whenever we make a page dirty. If the size exceeds a predefined constant number D', we choose one of dirty pages and remove it from the set (currently, we simply use FIFO strategy). We then protect the page again and scan marked objects in it to find unmarked direct children. While this technique incurs more overhead on write fault handler and subsequent writes, we can bound the size of non-trapped region to $O(R + D')$ rather than $O(R + M)$.

Determining the upper bound D' has an impact both on pause times and execution times. Although a small D' tends to reduce the amount of work in a quick termination check, too small D' would cause thrashing and increases protection costs. Section 5 shows performance of benchmarks when we bound dirty pages to sixteen pages.

4.2.2 Retrying Concurrent Marking Phases

Even if we limit dirty pages, we still cannot limit the amount of work in the final marking. The non-trapped area,

which has a bounded size, may have pointers to unmarked objects. It may be the root of a large linked data structure such as tree, whose elements may be all unmarked. In the basic algorithm, the final marking phase marks its all descendants atomically.

To avoid this problem, we *retry concurrent marking*. The basic strategy is to find such objects in the quick termination check and mark its descendants concurrently with the user program if it turned out to be a root of a big unmarked data structure. To achieve this, the termination check phase works as follows. It suspends all threads and scans non-trapped area (whose size is $\leq R + D'$). If it finds pointers to unmarked objects, it pushes them onto mark stack. And then the collector starts marking from detected objects, but stops marking if it takes a long time. To be precise, after scanning the non-trapped area, the collector marks no more than A bytes (A is 8K in our experiments). If the collector has finished marking before it reaches this limit, it starts concurrent sweeping phase. Otherwise, the termination check fails; it restarts concurrent marking phase. We let the remaining descendants of such white trees be marked in the subsequent concurrent marking phase. Thus concurrent marking and termination check phases may repeat several times. Each termination check incurs a bounded amount of work, which is $O(R + D' + A)$.

While we can bound the length of a single termination check, we do not have a theoretical bound on the number of repetitions. Instead, we adopt a strategy to reduce it as described in Section *4.2.3*. We change the default state of newly allocated objects during a collection cycle to reduce unmarked objects in the latter phases of a collection.

Boehm et al. have already introduced the idea of repeating concurrent marking. We consider our method differs from theirs in a subtle but an important way. In their method, the collector remembers the wall clock time when a termination check is started and starts marking from the non-trapped area, whose size is unbounded. If it took more than 50ms, the termination check aborts. In unfortunate cases, it may abort even before it finishes scanning the non-trapped area, thus *fail to find a pointer to a big data structure* while threads are suspended. After threads are resumed, they may move around the pointer to such objects, whereby "hiding" these objects from the collector during the concurrent marking. In this case, repeating concurrent marking makes no progress. On the other hand, our method guarantees the progress of collection, because termination check certainly visits *all* non-protected area and thus finds at least one pointer to unmarked objects if there is any.

4.2.3 Switching the Color of New Objects

By regarding new objects allocated during a GC cycle as unmarked, the collector can reclaim short-lived objects promptly. However, "allocating white" tends to repeat concurrent marking phases many times, because it produces many unmarked objects during each concurrent marking phase, some of which may be pointed to from the non-trapped area. If we regard them as marked, we can reduce the number of repetitions, but all objects allocated during a GC cycle are retained. We choose an intermediate strategy; we allocate objects unmarked during the first concurrent marking phase and marked in subsequent phases. This can reduce the number of repetitions, compared with the "always white" strategy.

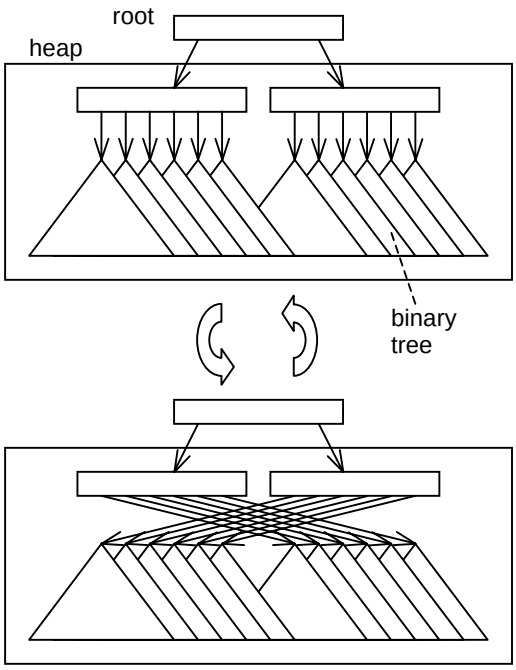

Figure 2: Data structures of `iukiller` benchmark. It repeats moving references to large binary trees many times in order to hide them from the concurrent collector.

5. EXPERIMENTAL RESULTS

We evaluate the performance of our collector on Sun Enterprise 4500 multiprocessor. This machine is equipped with fourteen 400MHz UltraSPARC processors. Its operating system is Solaris 8, and the page size is 8KB. We used several sequential and parallel programs written in C/C++ and examined the effects of the optimizations described in Section 4.

In the experiments, a GC cycle of concurrent collectors starts when the amount of free regions falls below 25% of the heap size. Concurrent marking phases make progress every 8K bytes of allocations. For the bounding dirty set optimization described in Section *4.2.1*, we let the upper bound of dirty set (D') be sixteen pages. For the retrying concurrent marking optimization described in Section *4.2.2*, each termination check marks no more than 8K bytes ($= A$) after scanning the root and dirty pages.

Newly allocated objects during GC cycles are treated as described in Section *4.2.3* unless explicitly mentioned. That is, we allocate objects unmarked before the first termination check and marked thereafter.

5.1 Results of a Synthetic Benchmark

This section demonstrates the effects of our pause time reduction optimizations using a synthetic benchmark named `iukiller`, which stands for "`i`ncremental `u`pdate `killer`." It frequently moves around references to large trees, so that concurrent marking tends to fail to trace them. Figure 2 shows how this benchmark works. First, it creates many binary trees whose depth is sixteen and stores references to roots of the trees into two arrays. Then it repeats swapping the contents of the two arrays, while allocating many short-

heap size (MB)	live data (MB)	GC algo-rithm	global pauses		local pauses	
			avg. (ms)	max. (ms)	avg. (ms)	max. (ms)
100	64	Stop	4119	4122	0.05	0.26
		Basic	917	2085	0.64	5.48
		BD	781	1802	0.65	5.20
		BD+R	3.59	7.46	0.97	11.7
200	128	Stop	8583	8607	0.05	0.34
		Basic	1734	4071	0.64	5.64
		BD	1548	3753	0.67	5.65
		BD+R	3.26	7.24	1.00	11.7
400	256	Stop	17023	17039	0.06	0.28
		Basic	3415	8205	0.63	5.95
		BD	3106	7166	0.66	6.10
		BD+R	3.57	7.42	1.00	11.2

Table 2: GC pause times in `iukiller` benchmark. "BD" and "R" stand for bounding dirty set and retrying concurrent marking, respectively. The "BD+R" collector achieves the maximum global pause times of <8ms.

lived objects so GC makes progress. We have measured the performance of this benchmark for three problem sizes. The heap size is fixed at 100M, 200M and 400M bytes.

Table 2 shows the pause times. The first and second columns show the heap size and the amount of live data and the third column the configuration of the collector. "Stop" stands for stop-and-parallel-mark collector, which is not concurrent. "Basic" resembles the basic concurrent collector by Boehm et al. described in Section 3, though it adopts the multiple collectors and the concurrent protection optimization to shorten the initial phase. "BD" is a concurrent collector that bounds dirty pages. "BD+R" adopts both bounding dirty pages and retrying concurrent marking. In the table, "global pauses" stand for pauses where all user threads are stopped; they occur in the initial and termination check phases (and in the final marking phase with the "Basic" and "BD" collectors). "Local pauses" are pauses where a single thread is stopped. They are due to protecting pages in the beginning of a GC cycle, marking, sweeping, and unprotecting pages in the write fault handler. Note that even "Stop" GC has local pauses, because its sweeping phase runs concurrently (incrementally) with the application.

The result indicates combining the two optimizations (BD and R) is essential to reduce pause times. The maximum global pauses of the "Basic" concurrent collector are 2085 to 8205ms. "BD" collector reduces them, but only slightly. They are still far from satisfactory for interactive or multimedia applications. To make matters worse, pauses get longer as heap and live data become larger. Only when we adopt both optimizations, are global pauses significantly reduced to 7.2 to 7.5ms for all the three heap sizes. The maximum local pause in the "BD+R" collector is sometimes longer than the global pause. The maximum local pause is dominated by the amount of objects marked at a time. Thus it will be reduced if we make the granularity of marking smaller.

Figure 3 shows the execution times of `iukiller` benchmark with a 100MB heap. It also shows the costs of collection. We observe all concurrent collectors incur more collection cost, and the execution times are 50–55% larger than that of the stop-and-mark collector. In this benchmark, the basic collector and optimized collectors are almost equal in the total execution times.

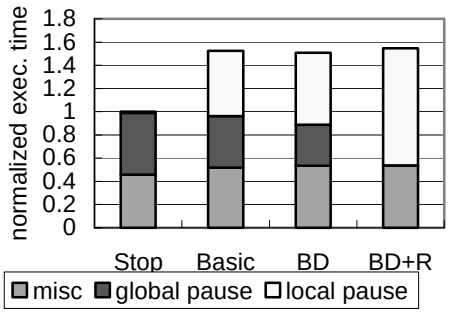

Figure 3: The execution time of `iukiller` benchmark with a 100MB heap, normalized to that of the "Stop" collector. "Misc" includes the execution time of the user program and the allocation time.

name	heap size	description
deltablue	25MB	an incremental constraint solver for acyclic constraint graphs [16]
espresso	10MB	a two-level logic optimizer for programmable logic arrays
N-Body	15MB(S) 40MB(P)	an N-body problem solver using the Barnes-Hut algorithm [3]
CKY	40MB(S) 50MB(P)	a context free grammar parser that allows ambiguous sentences[20]
cube	8MB(S) 20MB(P)	a puzzle solver that searches approximate solutions of the Rubik's cube

Table 3: Description of application benchmarks. "S" stands for heap size for sequential experiments. "P" is for parallel experiments.

5.2 Results of Sequential Programs

This section evaluates the collectors using five more practical programs, described in Table 3. Deltablue and espresso are widely used programs written in C. We slightly modified them to use our garbage collected allocator. Other three are C++ programs written by us, and also used in parallel experiments. During experiments, the heap size is fixed as shown in the table.

Figure 4 shows the pause times in the five benchmarks. It shows the average global pause, the maximum global pause, and the maximum local pause. We omitted the average local pause times, which were less than 1ms in all cases. The basic concurrent collector is successful in N-Body and cube; it reduces the maximum global pause times to one fourth of those with the stop-and-mark collector. It also reduces the maximum global pause times in deltablue and CKY, but not adequately; it is 78ms in deltablue and 165ms in CKY. In espresso, where living objects are small, GC time is short (about 10ms) even with the stop-and-mark collector. Bounding dirty pages ("BD") significantly reduces global

Program	avg. repetition	max. repetition
deltablue	2	2
espresso	1.6	5
N-Body	1.9	4
CKY	2	2
cube	1.1	2

Table 4: The number of repetitions of the termination check phases in the sequential benchmarks. In many cases the repetition finishes in a few times.

pause times of deltablue and CKY to 16ms and 9.6ms, respectively. Unlike `iukiller` benchmark in the previous section, BD alone has a significant effect. Yet, retrying concurrent marking has a substantial effect; the maximum pause times with all the optimizations ("BD+R") are now 2.6–4.5ms in the five benchmarks.

Figure 5 shows the total execution times of the benchmarks. We observe that the overhead of concurrent collectors is particularly large in CKY, mainly because of the overhead of local pauses. The total execution times of CKY are 53% longer than that with the stop-and-mark collector. In other four benchmarks, the overhead is 20% or less. In CKY, the difference between the basic collector and the optimized collector is also large. This is because "BD" optimization significantly increases the number of VM operations. We need more detailed investigation of the relation between the number of VM operations and the upper bound of dirty pages.

As described in Section *4.2.2*, our "BD+R" collector repeats the concurrent marking phases and the termination check phases. Table 4 shows the maximum number of repetitions in our five benchmarks. The average is no larger than two, thus our optimization that repeats the check rarely prevents progress of the applications. We observed, however, the termination checks are repeated four or five times in the worst case. We will investigate more sophisticated heuristics to reduce the repetitions in future.

5.3 Minimum Mutator Utilization

Although we have shown the pause times are significantly reduced above, small pause times alone are not enough to make a collector realtime. Even if each pause is small, tight clusters of small pauses might prevent applications from making "enough" progress. To quantify how realtime a GC is, Cheng et al. has described another metrics named *minimum mutator utilization (MMU)*. The mutator utilization of a given time window is defined to be the fraction of time allocated for the user program in that window. By measuring the mutator utilization for many time windows, we obtain the MMU for a given window size.

Figure 6 shows the MMU in the five application benchmarks. Each graph describes the relationship between window sizes (x-axis) and MMUs (y-axis). For relatively small window sizes, optimized concurrent collectors have much larger MMUs. MMUs with BD+R collector is 0.12–0.48 at granularity of 20ms, while it is zero with the stop collector in four programs out of five. In deltablue and CKY, optimized collectors show considerably better MMUs than the basic collector. MMUs of the stop-and-mark collector exceeds those of concurrent collectors in very large windows such as 100ms or larger, which merely indicates the overhead of the stop-and-mark collector is smaller than that of concurrent collectors.

5.4 Results of Parallel Programs

We use three application benchmarks, N-Body, CKY and Cube, for parallel experiments. While N-Body are written with a native thread library supplied by OS, CKY and Cube are parallelized using StackThreads/MP library [25], which schedule fine-grain user level threads on top of kernel threads. We gave a larger problem to N-Body and Cube than in the previous section to make them run sufficiently long. Each thread maintains its own dirty pages set, each of which is bounded by sixteen pages.

The machine used in the experiments was lightly loaded. We created no more than twelve kernel threads and bound each thread to a distinct processor, so we expect all the created threads are scheduled at all times (recall that the machine has fourteen processors).

Figure 7 shows pause times with four, eight, and twelve kernel threads. We observe concurrent collectors are useful to reduce global pause times in parallel applications too. Our optimizations reduce the global pause times by 34 to 83% compared to the basic collector. The difference in pauses between the stop-and-mark collector and concurrent collectors tends to become smaller as we use more processors, because pauses with the stop-and-mark collector are reduced by parallel marking.

The maximum global pause times of CKY are not reduced enough, which are 41ms, 23ms, and 33ms on four, eight, and twelve processors, respectively. We found that this is because StackThreads/MP uses very large execution stacks for this program. Since our current implementation never protects stacks, this increases the cost to scan roots in the termination check phases.

We observe that the local pause times of our collector sometimes get 10ms or longer. This can largely be adjusted by setting the granularity of marking. We have found another problem, however, which is the operating system's serialization of virtual memory operations. In the operating systems we tested, Solaris, Linux and IRIX, when a thread raises a write fault and tries to unprotect a page while another thread is protecting or unprotecting another page, the former thread is blocked despite these two threads are handling distinct pages. This makes the write fault handler take long time, increasing local pause times. The only essential solution for this problem seems changing implementation of memory management and `mprotect` system call [1].

Figure 8 shows execution times of the benchmarks normalized to that with the stop collector on four processors. Our collector imposes a large overhead in CKY, as in sequential experiments. In the three benchmarks, the total execution times with the BD+R collector are 3.9–48% longer than that with the stop collector, and up to 13% longer than that with the basic concurrent collector.

6. RELATED WORK

Incremental/concurrent mark-sweep collectors have been widely explored [11, 12, 13, 17, 27]. Most previous work

[1]While protecting pages inherently requires synchronization among processors, we may unprotect pages without synchronization by lazily synchronizing TLBs [1].

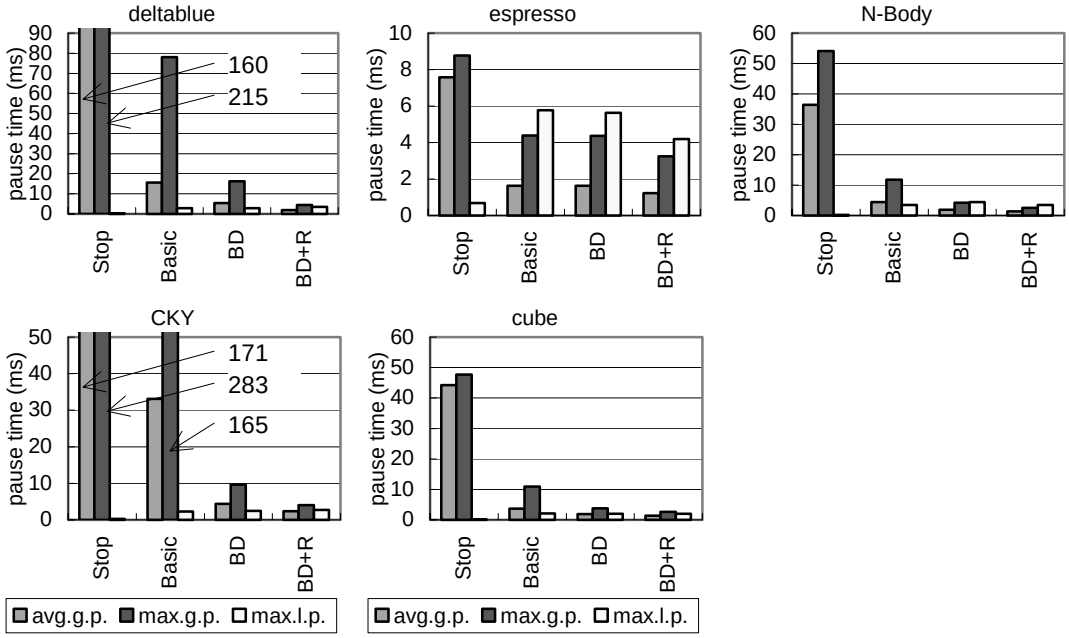

Figure 4: The pause time in the sequential benchmarks. The graphs show the average global pauses, the maximum global pauses, and the maximum local pauses for each collector. The "BD+R" collector achieves the maximum global pause times of <4.5ms.

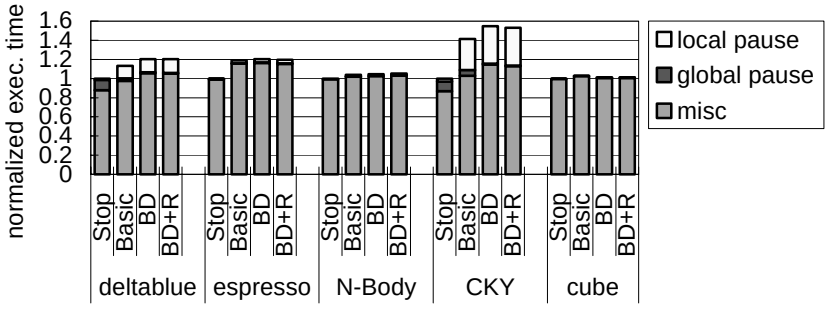

Figure 5: The execution times of the sequential benchmarks, normalized to that with the "Stop" collector. Each bar also shows the collection costs, which are broken down into global pauses and local pauses.

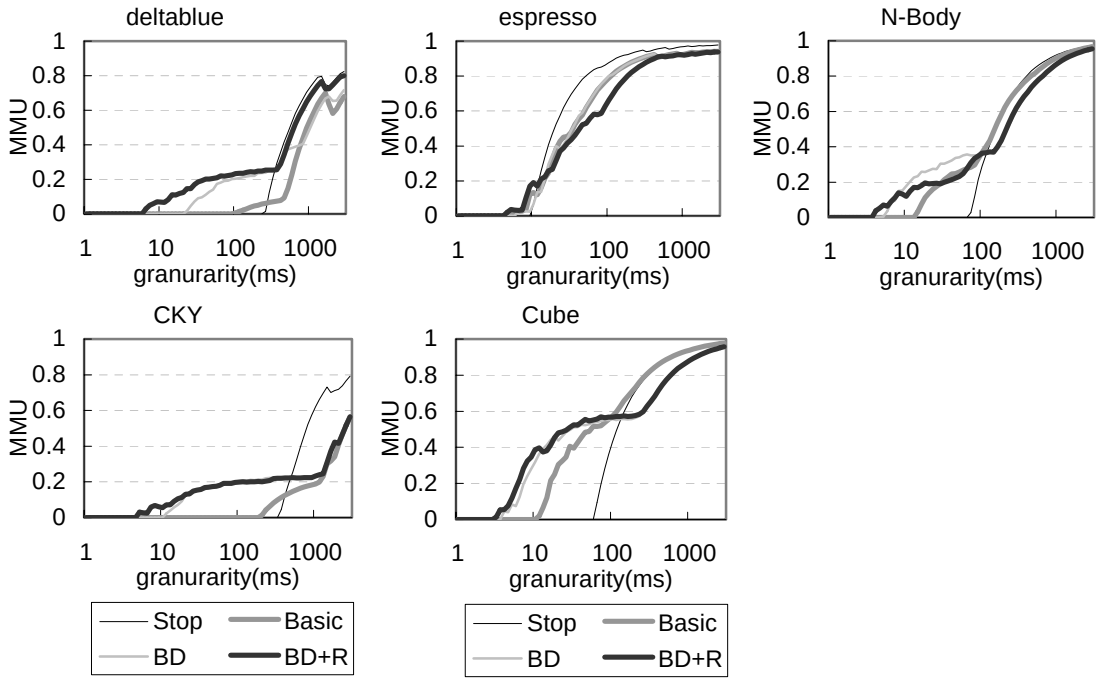

Figure 6: The minimum mutator utilizations (MMUs) in the sequential benchmarks. The mutator utilization (MU) for a given time window is the fraction of the user time in the window. The graphs show the minimum value for various window sizes. The "BD+R" optimized collector achieves higher MMUs for smaller window sizes.

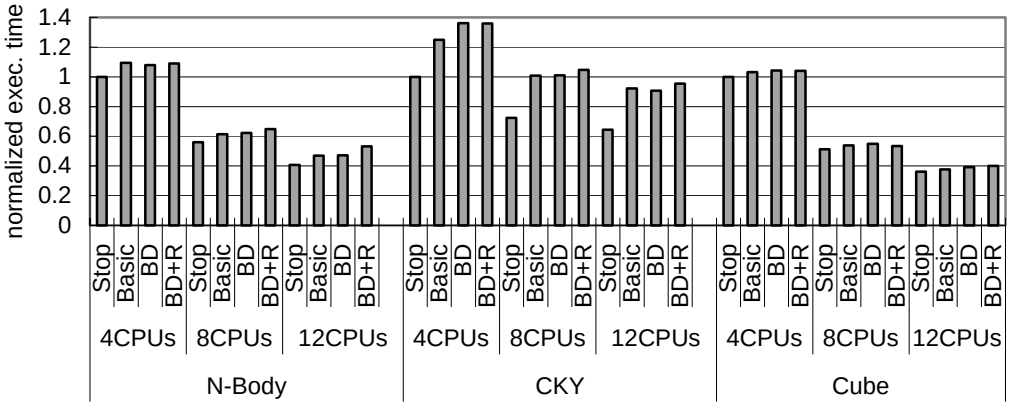

Figure 8: The execution times of the parallel benchmarks, normalized to that with the "Stop" collector on four processors.

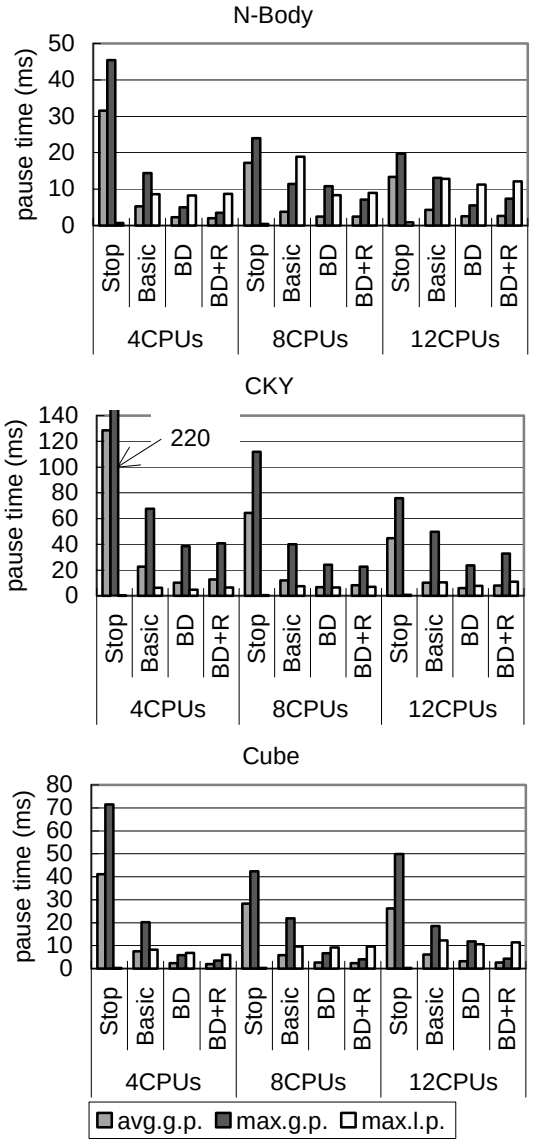

Figure 7: The pause times in the parallel benchmarks with four, eight and twelve processors. The graphs show the average global pause, the maximum global pause, and the maximum local pause for each collector.

assumes cooperation from compilers for read/write barriers. Our work focuses on the "conservative" approach that does not use cooperative compilers but relies on virtual memory primitives.

Appel et al. [2] described a concurrent copying collector relying only on virtual memory primitives. Their collector prohibits user threads from accessing from-space by a read barrier, which is much more expensive than a write barrier.

Furuso et al. [23] described a concurrent conservative mark-sweep collector based on a snapshot-at-the-beginning algorithm. It makes a virtual snapshot of the heap when a GC cycle starts. The problem of this approach is its memory overhead; the size of the snapshot may grow up to the heap size in the worst case.

As we have described in Section 3, our implementation is based on the collector by Boehm et al. [6], which is an incremental conservative collector. It uses an incremental update algorithm [11, 17] as its basis and implements a write barrier with virtual memory primitives. Their collector has already introduced the idea of retrying concurrent marking phases in somewhat primitive forms. However, because the number of repetitions is limited to two, the second final marking may be long. In our collector, the amount of work in a single termination check is always bounded. Although we have no theoretical bound on the number of repetitions, we reduce it as follows: (1) bound the number of dirty pages during concurrent marking phases, and (2) allocate new objects with marked from the middle of a collection cycle. The first optimization guarantees an amount of progress at every round of concurrent marking + termination check, and the second limits the amount of newly created unmarked objects. Together, they reduce the average number of retries while bounding the time of a single round.

Matsui et al. have proposed the complementary GC[22]. It performs an incremental update algorithm first. When a concurrent marking finishes, it then performs a snapshot-at-the-beginning algorithm instead of a final marking phase. This is a practical compromise between incremental update algorithms and snapshot-at-the-beginning algorithms; it allows some new objects allocated during a collection reclaimed at the end of that collection. It avoids the problem of incremental updates that prolongs the final marking phase by switching to the snapshot-at-the-beginning from the middle of a collection. It shares the potential problem of snapshot-at-the-beginning algorithms that the memory overhead for maintaining copies of updated pages can be very large (up to the heap size).

Printezis et al. [24] have described a generational GC for Java virtual machines. Their old generation collector is based on the algorithm by Boehm et al., though they utilize compiler support for write barriers. They adopt a "concurrent precleaning" technique, very similar to our bounding dirty pages. When the heap has too many dirty (grey) objects, some of them are scanned and made black. Unlike ours, their collector performs concurrent marking only once, and rescues unmarked reachable objects in the single final marking phase. They reported that with very large heaps, the maximum pause time exceeds 100ms. These results suggest the importance of our approach that repeats concurrent marking phases until the number of unmarked reachable objects gets small enough.

Cheng et al. [9] have implemented a concurrent parallel copying collector for multiprocessors. Like our collec-

tor, they achieve scalability by supporting multiple collector threads. Unlike ours, their collector uses compiler support for write barriers. When an object is updated, both the old referent and the new referent are made grey. Since it can ensure that all reachable objects have been traced when concurrent phase finishes, it needs no final tracing. As we have described, maintaining old referents is space consuming if we use virtual memory primitives, thus we did not take this approach. They describe techniques to reduce pause times such as dividing execution stacks into several parts to reduce times for scanning roots. The pause times of their collector are exceedingly short; they are 3 to 5ms at the maximum on their fifteen SML programs.

7. CONCLUSION

We have described an incremental update mark-sweep collector on top of virtual memory primitives provided by today's most operating systems. We argued that it is more difficult to reduce pause times in this setting than with compiler supports. In order to shorten the final marking phase that may cause unbounded pause times, we have described two main techniques: bounding the number of dirty pages and retrying concurrent marking phases. We do not insist that each technique itself is innovative, since similar techniques have been explored in different contexts. Instead, we have demonstrated that we can obtain remarkable effects by combining the two techniques through experiments. With basic incremental update algorithms, the pause times for final marking may be proportional to the heap size in the worst case. In contrast, our termination check in the optimized collector has an upper bound proportional to the number of dirty pages bounded by a predefined constant, plus the root size, which our current implementation does not attempt to bound but more involved implementations certainly can. We can reduce pause times to 4.5ms or less in all five application benchmarks we have tested. They are up to forty times shorter than those of the basic concurrent collector. The overhead that our method incurs was 9% or less to the basic collector. In addition to measuring pause times, we have shown our collector does not disturb application progress according to the minimum mutator utilization metrics.

We have shown that our collector also exhibits good performance for multithreaded parallel programs. It significantly reduces global pause times in our three parallel benchmarks. We have pointed out serialized `mprotect` system call seems to prolong pause times of individual threads.

We plan to make pause times even shorter by the following improvements. Current implementation does not protect stacks, so non-trapped area becomes potentially large if programs use large stacks. We have observed this situation in the parallel CKY benchmark. This can certainly be fixed by limiting the size of non-trapped part of the stack. While we have succeeded in reducing pause times, our collector incurs a larger overhead than the basic collector does. In the preliminary experiments, we observed the major reason is the increased protection operations to bound dirty pages. We will explore a better strategy to determine the bound, which is aware of both pause times and protection costs.

8. REFERENCES

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, 1991.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[3] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[4] Hans-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[5] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 93–100, 2002.

[6] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, 1991.

[7] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[8] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[9] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136, 2001.

[10] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, August 1993.

[11] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[12] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–83, 1994.

[13] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, January 1993.

[14] Toshio Endo. *Scalable Dynamic Memory Management Module on Shared Memory Multiprocessors*. PhD thesis, Department of Information Science, The University of Tokyo, September 2001.

[15] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of ACM/IEEE Conference on High Performance Networking and Computing (SC97)*, November 1997.

[16] Bjorn N. Freeman-Benson, John Maloney, and Alan

Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[17] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[18] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 7(4):501–538, 1985.

[19] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley & Sones, 1996. ISBN 0-471-94148-4.

[20] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, 1965.

[21] Yonezawa Laboratory. Parallel/distributed garbage collectors home page. http://web.yl.is.s.u-tokyo.ac.jp/gc/.

[22] Shogo Matsui, Yoshio Tanaka, and Masakazu Nakanishi. Complementary garbage collector. In *Proceedings of the 1995 SIGPLAN International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 163–178, 1995.

[23] Satoshi Matsuoka, Shin'ichi Furuso, and Akinori Yonezawa. A fast parallel conservative garbage collector for concurrent object-oriented systems. In *Proceedings of IEEE International Workshop on Object Orientation in Operating Systems*, pages 87–93, October 1991.

[24] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collection. In *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 143–154, October 2000.

[25] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71, May 1999.

[26] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, 1992.

[27] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.