Optimizations in a Private Nursery-based Garbage Collector

Todd A. Anderson

Programming Systems Lab
Intel Corporation
Hillsboro, OR 97124
todd.a.anderson@intel.com

Abstract

This paper describes a garbage collector designed around the use of permanent, private, thread-local nurseries and is principally oriented towards functional languages. We try to maximize the cache hit rate by having threads continually reuse their individual private nurseries. These private nurseries operate in such a way that they can be garbage collected independently of other threads, which creates low collection pause times. Objects which survive thread-local collections are moved to a mature generation that can be collected either concurrently or in a stop-the-world fashion. We describe several optimizations (including two dynamic control parameter adaptation schemes) related to garbage collecting the private nurseries and to our concurrent collector, some of which are made possible when the language provides mutability information. We tested our collector against six benchmarks and saw single-threaded performance improvements in the range of 5-74%. We also saw a 10x increase (for 24 processors) in scalability for one parallel benchmark that had previously been memory-bound.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.4.2 [*Operating System*]: Storage Management—Allocation/deallocation strategies, Garbage Collection

General Terms Algorithms, Management, Performance, Design

Keywords garbage collection; functional languages

1. Introduction

This paper describes a new garbage collector (GC) called TGC whose goal is to minimize memory-bus traffic by maximizing the cache hit rate and which is largely based on the Doligez-Leroy-Gonthier (DLG) collector [7, 8]. We aspired to this goal due to initial scalability results we obtained from our first benchmark (a raytracer) written in an experimental functional language designed for expressing parallelism. (Our language is similar to MIT's pH, parallel Haskell, and is explicitly parallel.) Initial scalability for the raytracer showed peak performance when only 4-6 out of 24 processors were in use. We conducted an analysis of the raytracer using Intel's EMON tool, which indicated that the memory bus was saturated. We further analyzed the raytracer with Intel's VTune Performance Analyzer [1] to observe the variation in the cycles-per-instruction (CPI) metric as additional processors were added. Since memory-induced throttling is reflected as higher CPI, CPI changes indicate which functions are memory-bound. This analysis revealed that most of the memory-bandwidth induced throttling was occurring in GC functions thus pointing at our original GC (call it JGC, a block-based, stop-the-world mark-sweep-compact GC heavily influenced by [13, 14] that was originally optimized for Java and relatively low numbers of threads) as a target for replacement. Thus, the primary goal for our new GC was to maximize cache re-use in order to minimize memory traffic across the bus, thus increasing the number of threads that could run simultaneously without memory-induced throttling. However, even for single-threaded applications, maximization of the cache hit rate is likely to be a good strategy.

Most high-performance GCs (including JGC) use a thread-local allocation area (nursery) from which a thread can allocate without having to acquire a lock. When that nursery is full, the thread will request a new nursery from the GC. Our VTune analysis showed that much of the memory traffic on our multi-processor system was caused by threads zeroing these new nurseries. Thus, we sought to identify why these new nurseries were not in the cache and how to address this situation. When the GC no longer has any free nurseries to give out, a garbage collection is triggered. In a multi-processor system, it will often be the case that the processor performing the collection of a former nursery will be different from the (one or more) processors used by the thread which filled that nursery. Upon completion of the GC, the block of memory will again be given to what is likely a different thread running on a different processor. Thus, nursery blocks frequently cycle between predominant access from one processor to another for two reasons: 1) predominant access cycles from one thread to another and 2) thread migration between processors. Each such predominant access transition likely requires the block of memory to pass over the bus to reach cache on a different processor.

In TGC, we try to minimize memory-bus traffic and nursery ping-ponging by assigning each thread a permanent nursery that is repeatedly used then collected by the same thread and by pinning threads to individual processors. (The compiler for our language extracts fine-grained parallelism from the program and places those work items on load-balancing work queues that are then serviced by one pinned thread per processor.) The contributions of our paper include:

- The ability for both mutable and immutable objects to be stored in a thread's permanent, private nursery and optimizations related to those private nurseries. These optimizations include:
  1. Dynamically estimating the optimal private nursery size.
  2. Speeding up stack walking for root set enumeration.
  3. Optimizing the timing of private nursery collections.
- Two concurrent collector optimizations:

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

1. Leveraging mutability information to allow a concurrent collector to compact most of the heap while avoiding the need to atomically flip pointers.
2. Running the concurrent GC thread only enough to keep up with mutator demand.

2. TGC Design and Implementation

TGC consists of a shared public heap, managed as a set of blocks, and one private nursery (PN) per thread (see Figure 1). This organization is similar to the one described in [8]. PNs are collected via a mark-copy algorithm. Each thread also has a transient, public heap nursery into which surviving PN objects are copied lock-free but which is accessible by any thread. When a public heap nursery becomes full, the GC gives the thread a different nursery. The public heap is collected via a mark-sweep-compact algorithm (Section 2.2).

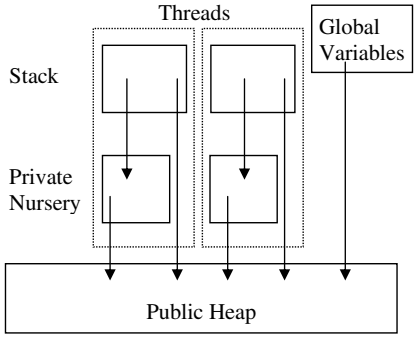

Figure 1. Memory Organization

2.1 Private-Nurseries

A PN is a permanently thread-local memory area into which a thread allocates objects. Each PN maintains the PN-invariant, which states that pointers into a thread's PN can only come from objects in the same PN or from roots on the thread's stack (not from the public heap, globals or from other threads' stacks). This invariant implies that threads cannot access objects in other threads' PNs and it enables independent collection, i.e., any number of threads may be simultaneously collecting their PNs without interfering with or requiring collaboration from any other thread.

When a PN cannot satisfy an object allocation request due to the PN lacking space to hold the object, a garbage collection of the PN is triggered (a PN-collection). Since no other thread or global can contain a pointer into the PN, the PN-collection must only find the thread's stack roots and from those compute the transitive closure of reachable objects in the PN. Each reachable object is then copied into the thread's public heap nursery. PN-object pointers in the copied objects are then updated to those objects' new public heap locations. Then, stack pointers to PN objects are similarly updated. Finally, the used portion of the PN is zeroed and new objects are again allocated at the beginning of the PN. Most objects in our benchmarks die quickly and thus PN-collections reclaim the memory used by the vast majority of allocated objects. In some of our benchmarks up to 99.8% of objects become unreachable (0.2% "survival rate") while in a PN.

The most common trigger of PN-collections, for most applications, is an attempt to violate the PN-invariant. If a thread were able to write a pointer to a PN-object into an object in the public heap (or a global), the PN-invariant would be violated. To prevent this situation, in our system, our code generator emits calls to TGC's write barrier when a pointer field is updated. If the write barrier detects a write of an object in a PN into a slot not inside the PN or the thread's stack, that would violate the PN-invariant and so the write barrier triggers a PN-collection. At the end of this collection, what was formerly a pointer to a PN-object will have become a pointer to that object's new location in the public heap. The write can then be performed without violating the PN-invariant.

While most objects are allocated in the PN, objects larger than the PN must be allocated in the public heap. Moreover, it makes little sense to allocate objects whose size is even a large proportion of the PN size in the PN since such a large allocation will mean a PN-collection will likely happen soon which will just require the object to be copied into the public heap anyway. As such, we chose to allocate objects that would take up more than 40% of the PN space directly in the public heap. This is a tradeoff since pretenuring that is too aggressive may cause excessive PN-invariant collections since large objects tend to be arrays of pointers.

2.1.1 Write Barrier Implementation

TGC's write barrier (WB) is implemented in assembly code. This code does not modify any callee-saved registers so no prolog/epilog sequence is used. In our system, we reserve ebx to continually hold a pointer to the current thread's data structure. The first part of the thread data structure is a pointer to the thread's PN data structure. The first entry in the PN data structure is the starting address of the PN. Thus, the WB starts by twice dereferencing ebx to get the start of the PN which is then subtracted using unsigned arithmetic from the address of the slot (passed as a parameter) into which a value (also a parameter) is being written. If this result (interpreted as an unsigned number) is less than the private nursery size then the write is to a slot in the private nursery and the WB jumps to a code segment that writes the value into the slot and returns. Thus, the WB is optimized for the common case (over 99% in most of our benchmarks) in which the slot is in the PN and a simple write is all that is required. Otherwise, a PN-collection is triggered (and the value parameter enumerated as a root) and then the same code segment as above is used to write the new value into the slot. Both the thread and PN data structures tend to stay in the cache so both dereferences in the WB tend to be cache hits. Modifications to this WB to support optimizations are described in Sections 2.2.2 and 3.1. By optimizing for the common case and eliminating prolog overhead, our write barrier is relatively inexpensive as it typically adds less than 3% overhead to most programs.

2.1.2 Dynamic PN Sizing

The goal of selecting a PN size is to have the highest performing mix of PN and non-PN data in cache (typically L2). In TGC, one can specify the PN size manually or use a dynamic scheme. Our existing benchmarks performed best using a fixed-sized PN due to the initial period of sub-optimality with the dynamic scheme and slightly sub-optimal selection of PN-sizes. However, we predict that this dynamic scheme may outperform any single fixed PN-size for applications exhibiting phase behavior or for applications whose threads have distinct optimum PN-sizes, since the algorithm runs independently for each thread.

Our dynamic PN-size adaption algorithm uses normalized PN-collection times as a proxy for cache performance and works similar to a binary search. PN-collection times are normalized by dividing the PN-collection time by the amount of data allocated in the PN since the last PN-collection. This quotient is called the time-per-byte metric (TPBM). If a PN size is too large then portions of the PN will be evicted from the cache to satisfy data accesses from the mutator. Zeroing these evicted portions during the next PN-collection will cause cache misses which will increase the TPBM beyond what it would have been had the entire PN stayed in the cache. Conversely, if the PN size is too small then fewer objects will have died (a higher survival rate meaning more marking and

moving work) and PN-collection overheads will be a larger proportion of PN-collection time (for a fixed amount of data). We believe these two factors help apply opposing pressure and cause the following algorithm to converge on an approximately ideal PN size.

Let TPBM(A) be the average TPBM for a given sample A collected over several PN-collections during which the PN size in use is denoted by SIZE(A). First, a sample TPBM(S_1) is collected for SIZE(S_1) equal to the L2 cache size by artificially restricting the available space in the PN to the specified amount. Then, the algorithm decreases the PN size by half ((S_2) == SIZE(S_1)/2) and collects the next sample TPBM(S_2). If TPBM(S_2) is less than TPBM(S_1) then a sample S_3 is taken, again for one half of S_2's PN-size value. This process is repeated until some TPBM(S_N) is taken which is larger than TPBM(S_{N-1}). At this point, TPBM(S_N) $>$ TPBM(S_{N-1}) and TPBM(S_{N-1}) $<$ TPBM(S_{N-2}). The minimum TPBM is now between S_N (call it bottom) and S_{N-1} (call it middle) or between S_{N-1} and S_{N-2} (call it top).

We next take two samples where the PN-size of the first sample X is half way between the bottom (SIZE(S_N)) and the middle (SIZE(S_{N-1})) and the second sample Y is half way between the middle (SIZE(S_{N-1})) and the top (SIZE(S_{N-2})). If TPBM(X) is less than the middle (TPBM(S_{N-1})) then the process is recursively repeated with the new "middle" being X, the bottom sample remaining unchanged and the top sample becoming N-1. If TPBM(Y) is less than the middle (TPBM(S_{N-1})) then the process is repeated with the new middle being Y, top unchanged, and bottom equal to N-1. If neither TPBM(X) or TPBM(Y) is less than TPBM(S_{N-1}) then the X becomes the new bottom and Y the new top and the process described in this paragraph is repeated. As very small differences in PN sizes are unlikely to cause much of an effect, we further introduce a threshold to terminate this recursive process such that if the difference between SIZE(TOP) and SIZE(BOTTOM) is less than some small percentage the process terminates. When this process terminates, SIZE(middle) is used as the PN size.

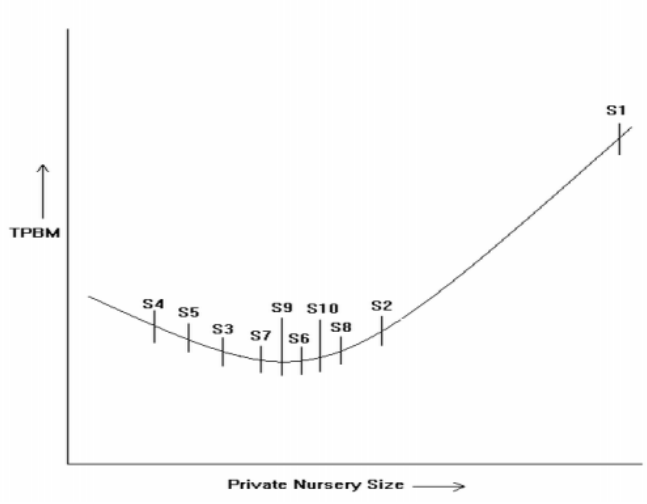

Figure 2. Dynamic Algorithm Example

Figure 2 shows an example of the dynamic algorithm for one possible PN-size/TPBM curve and the point at which each sample is taken. Samples S_1 through S_4 are collected by halving the previous PN size until sample S_4 is found whose TPBM is greater than the previous sample. Samples S_6 and S_5 are the midpoints of the previous three samples with S_6 now having the lowest observed TPBM. Samples S_7 and S_8 are the midpoints of S_3, S_6 and S_2 and S_6 continues to have the lowest TPBM. Samples S_9 and S_{10} are the midpoints of S_7, S_6 and S_8 and S_9 now has the lowest TPBM. For this example, we assume the difference between S_7 and S_6 is below the threshold and the algorithm terminates.

While most benchmarks we have examined seem to prefer a PN-size ranging from 128K to 500K, some benchmarks prefer a PN-size larger than the L2 size. As such, if TPBM(S_2) is greater than TPBM(S_1), the previous process is performed except that each successive sample will have twice the PN-size of its predecessor (rather than half) and the algorithm will search for the optimal in the same way described in the previous paragraphs.

Once this algorithm terminates, TGC continues to collect TPBM samples of PN-collections. If the weighted moving average of these TPBMs diverges from the TPBM of the previously computed optimal sample by more than some fixed percentage then a phase shift in the application may have occurred. As such, a new binary search is started where SIZE(S1) is equal to the current PN-size and each subsequent sample's PN-size is 90% of the previous one (rather than 50% as in the first iteration of the algorithm). Reducing the difference between sample sizes accounts for the fact that the new optimum would in many cases be close to old optimum and restarting the process with SIZE(S1) equal to the L2 size could introduce a period of substantial non-optimality.

2.1.3 Reducing PN-Collection Overhead

With the introduction of PNs, rather than having a small number of major GCs, a large number of PN-collections are executed. The number of PN-collections can be 100,000 times greater than the number of GCs in JGC. Having this many small collections can place stress on parts of a system that often are not very optimized. Indeed, the original tests with TGC revealed excessive PN-collection overhead associated with stackwalking. Straight-forward optimizations (such as caching various information) of the stackwalking code reduced this overhead by about 40%. We also developed a watermark mechanism that further reduced the original stackwalking overhead by 50% by reducing the number of frames that need to be walked.

If a frame is on the stack when a PN-collection occurs, the PN-collection will update any pointers in that frame to point to the new public heap versions of those objects. If the same frame is still on the stack when the next PN-collection occurs and no execution has taken place within that frame then it is not necessary to enumerate roots within that frame as they cannot point into the PN. (Another requirement for this optimization is that one knows that object references in frames are not indirectly modified via pointers to those references passed to other functions. Our functional language guarantees this requirement.) This observation led to the implementation of a watermarking mechanism to differentiate the portion of the stack containing frames that cannot contain PN references from those that could. Conceptually, when a PN-collection processes a frame for roots, it puts a watermark on that frame. When a PN-collection is about to process a frame for roots, it checks to see if the frame is watermarked and if so the PN-collection terminates stackwalking. If a given frame is watermarked then every frame older than the given frame must also be watermarked.

When the GC watermarks a frame, the system copies the real return address of the frame into the jump address of a free watermark stub. Then, the system overwrites the return address in the frame [4, 5, 15, 19] with the address of the watermark stub. Thus, when the return instruction is executed for this frame, the watermark stub is invoked. Each watermark stub is seven assembly language instructions that increments the top of the watermark stack (see below), pushes the current stub's address onto that stack and then jumps to the real return IP for the frame.

Each thread has a region of memory in which watermark stubs are created. To determine if a frame is watermarked, the system inspects the frame to see if the return address is in the memory

range containing the watermark stubs for the thread. If the frame is watermarked and the system needs the real return IP (e.g., to determine the set of live references at that point), the real return IP can be found at a fixed offset from the start of the watermark stub (i.e., the target of the jump address). For deeply recursive stacks, if the initial allocation of watermark stubs is insufficient then additional watermark stub regions can be created. The frame watermark check would then compare the return address against each of the extant watermark regions.

Each thread also has a watermark stack which records the addresses of free watermark stubs. When a frame is being watermarked, a pop is performed on the watermark stack to get the address of a free watermark stub. When run, a watermark stub pushes its own address onto the watermark stack to signify that the watermark stub is no longer in use. An illustration of a stack with Frame N+1 watermarked is shown in Figure 3.

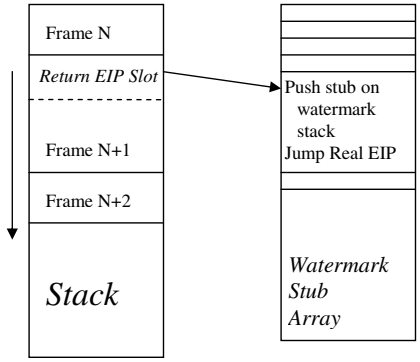

Figure 3. The Frame Watermarking Subsystem

2.1.4 Discussion

We have collected statistics on the frequency and location of write barriers that trigger PN-collections and the percentage of data surviving those collections. In our benchmarks, the distribution of PN-collections across write barriers appears to obey a power-law relationship. Moreover, PN-invariant collections have the tendency to have a lower percentage of surviving objects than capacity PN-collections. We found the cause of this tendency by inspecting the write barriers causing PN-invariant collections and determining that they tended to occur at (sub)computation boundaries where more temporary objects are dead whereas capacity PN-collections occur randomly within the context of a computation.

We tried introducing *preemptive PN-collections* to avoid the higher survival rates of capacity PN-collections. Preemptive PN-collections places a check in the fast-path of the write barrier to trigger a PN-collection if the amount of used space in the PN exceeds some threshold percentage of the PN size (we tried 85%-98%). Using this approach, we found that capacity collections were eliminated and the preemptive PN-collections that replaced them were faster and had lower survival rates. However, the added cost of the threshold check in the write barrier balanced out the gains from faster collections and so no net gain in performance was seen. In future research, we plan to investigate if it is possible to more cheaply perform this threshold test.

2.2 Major GCs

In TGC, we call a collection of the public heap a major GC. When using PNs, the number of non-concurrent major GCs drops by an amount equal to the inverse of the PN survival rate (as in Section 2.1 up to 99.8% reduction in major GCs). Thus, in many of our benchmarks, no major GCs are observed. However, longer running applications will eventually need a major GC to reclaim space in the public heap. TGC can be run in two modes providing two types of major GCs, concurrent and non-concurrent. Non-concurrent major GCs tend to minimize certain types of overhead and are typically best for throughput-oriented applications. Our non-concurrent major GC is a mark-sweep-compact (roughly 1/16 of the heap is compacted per GC, the rest is merely swept), stop-the-world GC (pause times in the tens of milliseconds) in which each stack, PN, and global variable is scanned for pointers into the public heap, which act as roots for the computation of the transitive closure of reachable objects in the public heap. While non-concurrent, each phase of this collector is done in parallel. Our concurrent GC is a mark-sweep (non-moving although there is an exception as noted in Section 3.2) collector using a tri-color scheme[6] similar to the one in [8] and [7]. Our concurrent collector has no stop-the-world phase and is suitable for applications whose threads have soft real-time requirements. The average pause time for threads under the concurrent GC is less than 10us.

2.2.1 The Concurrent Collector

The concurrent collector itself may be run in two modes: continual and minimal. In the former, a GC thread continually performs concurrent GCs. When one GC is finished, the GC thread immediately starts the next. This minimizes the possibility that a mutator asks for a new public heap nursery and none is available but increases the overhead to mutators who have to do extra work when the concurrent GC is active. Moreover, time spent by the concurrent GC thread performing GCs that are not strictly necessary is time not spent performing useful (mutator) work.

In minimal concurrent mode, the concurrent GC thread awaits a signal from the GC's nursery allocator before starting a GC. TGC's nursery allocator knows how many public heap blocks are free and can become nurseries. Furthermore, the nursery allocator observes the rate at which nurseries are being requested by keeping a weighted moving average of the inter-arrival time between nursery requests. At each nursery allocation, TGC divides the number of free blocks by the block allocation rate estimate yielding an estimate for when block exhaustion will occur. If this estimate is less than the maximum time to complete a concurrent GC then TGC sends a signal to the concurrent GC thread to start a GC.

Since a concurrent GC started too late (which can result in threads exceeding their pause time guarantees) is much worse than one started too early (a little more cumulative GC overhead), TGC errs on the side of starting the concurrent GC earlier than needed by pretending the public heap has fewer free blocks than it really does (by 1-2% of the total number of blocks) and by multiplying the true maximum observed concurrent GC time by some small multiplier (typically around 2). This approach accommodates the occasional GC that is longer than any previously seen and provides an extra buffer of blocks. In non-concurrent mode, all swept blocks are made available for reuse as nurseries at the end of the GC. In concurrent mode, this approach can lead to a situation where a thread requires a new nursery and while the concurrent GC thread has swept a number of blocks that could be used as nurseries those have not been officially returned to the nursery pool and so the thread has to wait and subsequently exceed its maximum pause time. Therefore, in concurrent mode, when a block is swept, the GC thread checks if the number of available nurseries is near zero and if so it immediately returns all previously swept blocks to the nursery pool. We do not return swept blocks immediately to the nursery pool as a matter of course because doing so requires acquiring a mutex and we seek to minimize this locking by batching block returns to the nursery pool as much as possible. The interaction between the block allocation rate estimator and the likely cost of

the next GC is complex and further refinement of this approach is an area of future investigation.

Nevertheless, we have observed an order of magnitude reduction, for some benchmarks, in the number of concurrent GCs when using minimal mode compared to continual mode, even at 24 processors, all the while staying within our maximum pause time requirements. The percentage of time the concurrent GC must run increases as you add processors but since PN-collections absorb so much of the collection work, only as little as 10% of one processor was needed for concurrent GC even up to 24 processors. The remaining 90% of that processor can be used by a mutator to do useful work. Indirectly, the mutators also run faster in minimal concurrent-GC mode as the mutators do not have to spend time coloring objects if no concurrent GC is in progress. While the concurrent marking and sweeping phases could be parallelized, we have not done so because a single concurrent collector thread has been sufficient to keep up with mutator demand.

2.2.2 Concurrent Collector Algorithm

The concurrent collector can be in one of three states, idle, marking, and sweeping. No work is done in the idle state. In the marking phase, all reachable objects are marked black according to the tricolor marking scheme in [7, 8]. In sweeping mode, public heap blocks are scanned and the space used by white (non-reachable) objects reclaimed.

The marking phase starts by setting a global flag to indicate that the concurrent GC would like to enter marking mode. The first task in marking mode is to enumerate pointers (roots) to objects in the public heap present in threads' stacks and PNs and to mark those objects gray (reachable but not yet processed). To make concurrent minimal mode (Section 2.2.1) function better by increasing the predictability of concurrent GC times, we do not artificially wait for threads to enumerate themselves during a PN-collection (as described in [8]) as we found this added significant time variability. Instead, the concurrent GC thread stops each thread one-at-a-time, cross-enumerates its stack and PN for roots into the public heap using a subset of the normal PN-collection code, sets the thread-local marking mode flag, and then restarts the thread. This process of enumerating roots and restarting the thread is usually less than 10us. PN-collections also inspect the global flag and will self-enumerate if the concurrent GC thread has not yet enumerated the given thread.

When a thread is stopped for cross-enumeration, occasionally that thread will be stopped in a PN-collection at the point in which it attempts to allocate objects in the public heap (no other stopping point is possible in a PN-collection). If this is the case, it is not necessary to re-run the full subset of the PN-collection code to walk the stack and find public heap pointers in the PN as the PN-collection maintains data structures with this information. As an optimization, PN-collections do not typically record pointers to public-heap objects. As such, the concurrent collector takes the set of live objects and re-scans those for public-heap pointers.

During the marking phase, all reachable objects must become black. Each thread gives the concurrent GC thread a snapshot of its roots at one point in time but then continues creating additional objects and moving those to the public heap. As such, threads must keep the concurrent GC thread apprised of these additional reachable objects. During a PN-collection, the thread-local marking mode flag is checked. If the flag is set, the PN-collection marks the objects black as they are added to the public heap (no need to mark the objects gray since any references to other PN objects will also be moved by this process and marked black).

After receiving the root snapshot from each thread, the concurrent GC thread starts to process gray objects. A gray object, G, is taken from the gray object data structure and processed for pointers to objects into the public heap. Those objects are then marked as gray (and therefore added to the gray object data structure) and then G is marked black. This process is repeated until there are no more gray objects. At that time, the concurrent GC scans for additional gray objects created by mutators via the write barrier. In concurrent mode, the write barrier inspects the thread local marking mode flag when a public heap slot is being written. As in [8], the old and new pointers into that slot are colored gray by adding them to a thread-local gray list. The concurrent thread repeatedly cycles through these thread-local gray lists one at a time adding gray objects from those lists to its own data structure and processing those objects as above. Once the concurrent thread completes one cycle through all the threads without finding additional gray objects then all reachable objects are black and the concurrent thread transitions to sweeping mode.

To enter sweeping mode, the GC thread modifies its global mode flag to indicate it wishes to enter sweeping mode and then sets each thread's local sweeping mode flag. The concurrent GC thread then begins to scan each public heap block in ascending memory order. As black objects are found in a block, those objects are unmarked. The space left by white objects is reclaimed for subsequent use. As each new block is scanned, a global sweep pointer is updated to contain the address of the start of that block.

When a PN-collection moves an object to the public heap and the thread-local sweeping flag is set, the PN-collector compares the address of the new public heap object with that of the global sweep pointer. If the new object has a higher memory address than the global sweep pointer, the PN-collector marks the new object black (in order to keep it alive when the concurrent collector soon scans the block that the new object is in) and places the object on a post-sweep scan list. Since access to the global sweep pointer is not atomic, it is possible for a thread's PN-collector to believe a new public heap object is greater than the sweep pointer when in reality the sweep pointer was recently updated to a value higher than the new public heap object. This can lead to objects being marked lower than the true sweep pointer. Unless those objects are later unmarked, the next GC cycle will believe those objects have already been traced and will not re-scan them which can lead to reachable objects being reclaimed. To prevent this, after the concurrent GC thread completes its sweep, it sets the global idle mode flag and each thread's local idle mode flag (so no more objects will be added to the post sweep scan list) and then removes objects from the post-sweep scan list and makes sure that they are unmarked.

3. Optimization with Mutability

Some languages, particularly functional languages, can provide information to the GC about whether an object is mutable (may be modified in the future) or immutable (object will not be modified in the future). In [7, 8], mutable and immutable objects are differentiated at compile time and mutable objects allocated in the public heap. Our system can also identify many objects as immutable at compile time but we have noted that many mutable objects spend only a short amount of time being mutable and thereafter are immutable. In our system, the code generator emits code to change the vtable of an object at the point at which it determines that that object transitions from mutable to immutable. Thus, for all types that start as mutable, our system allocates two vtables for that type, one each for the mutable and immutable states. Therefore, unlike [8], we allocate both mutable and immutable objects in the PN. Many of the mutable objects later become immutable and can therefore take advantage of the optimizations described in this section. In all other parts of the system, the mutable and immutable vtables for a given type are considered identical.

3.1 Write-Barrier Optimization

In some applications, such as our raytracer, we observed that the application seemed to be doing an excessive number of PN-collections caused by attempts to violate the PN-invariant. Subsequent analysis showed that the pixel array was stored in the public heap and that every time a completed pixel (stored in the PN) was written into the pixel array, the write barrier was detecting a violation of the PN-invariant and triggering a collection. Moreover, it was noted that the pixel objects were all small and immutable.

If the write-barrier detects an attempt to write a pointer to an object A that would violate the PN-invariant, the write barrier determines if object A is immutable and contains no pointers to other objects. If so, the write-barrier creates a copy of object A in the public heap and then completes the write by using the new public heap copy of A. Thus, more than one copy of the object exists and so long as pointer comparison is not used to test equality, this optimization can be used. This optimization reduced the number of PN-collections by up to 90%. A form of this optimization is also employed in [8] but we can also apply the optimization to objects that were allocated as mutable and later transitioned to immutable.

We explored several variations on this policy. First, the policy as described above allows an unlimited number of copies of an object to be created. Moreover, with no record of an object being copied, at the next PN-collection (if a copied object is still alive) it will again be copied into the public heap. In principle, this could cause the public heap space to fill up more quickly thus causing additional major GCs. We performed a test in which we limited the number of copies that could be created in the public heap to one by maintaining an auxiliary thread-local data structure (and suppressing the additional copy during the next PN-collection). This structure mapped the PN location of an object to its public heap copy. In practice, we found that the overhead of maintaining this data structure and checking it (in each write barrier which violated the PN-invariant) exceeded any benefit from reducing multiple copies. Another option is to add a forwarding slot to each object to contain a pointer to the object's public heap copy. Likewise, any benefit from reducing multiple copies was overwhelmed by the cost of increasing the size of every object.

Second, we tried allowing object A to contain pointers to other objects. In this approach, it is necessary to examine the transitive closure of objects from A and confirm that all such objects are also immutable. If all reachable objects are immutable then copies of each such object are made in the public heap with pointers in those copies updated to point to the new public heap versions. We also tested this approach and found that it did not improve performance in our benchmarks for two main reasons: 1) a significant amount of time was wasted computing transitive closures containing mutable objects which invalidate the optimization, and 2) the base optimization (i.e., immutable, pointer-less) already pushes the PN-size at collection past the inflection-point discussed in Section 4. The cost of copying large trees of objects can also begin to approach the cost of a PN-collection with the additional downside that the tree will be copied again during the next PN-collection.

3.2 Moving Concurrent Collector

The non-moving concurrent collector described in Section 2.2.1 has the potential to fragment the public heap and prevent large object allocations from finding sufficient contiguous space. When the concurrent collector is used and the language is able to flag objects as immutable, each public heap block is tagged as either containing mutable objects or immutable objects. Then, each thread has a mutable public heap nursery and an immutable public heap nursery. PN-collections move objects to the correct public heap block based on the object's current mutability status. Subsequently, in each concurrent GC cycle, some revolving portion (about 1/16) of immutable public heap blocks have their objects compacted. Again, since the objects are immutable and equality is not done by pointer comparison, it is possible to have multiple extant versions of an object.

When an immutable public heap block that is scheduled to be compacted is being swept by the concurrent GC, the objects in the block are copied/compacted to another immutable public heap block and an entry is recorded in an auxiliary "moved table" with the old location, the new location of the object, and the number of the GC in which the entry was recorded. The new objects are also marked black if the new location is higher than the current block being swept (so that the subsequent sweep of the destination block will identify the new locations as live objects). A remembered set is kept of pointers from objects in immutable blocks into compacted blocks so that after the sweep phase those pointers can be updated to the new location of the moved objects (using the moved table). This remembered set also includes pointers in the old and new versions of the moved objects themselves. Thus, most public heap pointers are updated to their new value in the same GC cycle.

In subsequent GC cycles, as a pointer into the public heap from the stack, from objects in the PNs, or from globals is enumerated to the GC, the GC consults the "moved table" to determine if that object has been moved in a previous GC cycle. If so, the pointer is updated to the new location. Thus, most non-public-heap pointers to moved objects are updated during the GC cycle subsequent to when they were moved.

The remembered set of slots to compacted objects does not encompass those slots (containing pointers to compacted objects) in mutable objects. Without a global stop-the-world phase, it is not possible to safely overwrite a pointer to a compacted object in a mutable object slot as there may be a race with a mutator thread to modify the same slot. Given that some mutable objects are transitioned to immutable after entering the public heap, we allow objects listed in the moved table to exist there for several GC cycles to give the opportunity for mutable objects pointing to those objects to become immutable and update their pointers. However, some objects never transition to immutable so we set a limit for the number of GC cycles that an object can be listed in the moved table (so as not to over-populate the moved table with useless entries).

At the end of each concurrent GC, we scan the moved table and remove entries from that table if the entry satisfies one of two criteria. First, we remove the entry if the original object location is not marked. If the original location is not marked then there were no pointers seen to the old version of the object and thus no thread can potentially use this entry in the future so it is safe to remove the entry. Second, we remove the entry if the current GC number is a fixed amount greater than the GC number in which the entry was originally added. In our experience, the number of mutable objects that become immutable and then use the moved table to update a pointer drops off rapidly with each additional GC. We found little benefit with extending the number of GCs a moved table entry can exist past five. Finally, at the beginning of each concurrent GC mark phase, the new versions of objects in the moved table are kept alive by coloring them gray.

In most of our benchmarks, upwards of 90% of objects are immutable at the time they leave a PN. Moreover, the objects that are mutable in our system tend to be small with a low variance in size. Thus, mutable blocks may be fragmented but this fragmentation does not have much effect on mutable object allocation. To measure fragmentation, we count the number of gaps between objects (representing dead objects) in all the public heap blocks. When compaction of immutable public heap blocks is enabled, we consistently see reductions in the amount of fragmentation by 95-99%.

4. Object Survivability

For most applications in our functional language, object survivability drops off sharply as a function of time. There is typically some inflection point in object lifetimes before which most objects are still alive and after which a large percentage of objects become unreachable. If PN-collections are occurring so frequently that objects cannot pass this inflection point then PN-collections can become very expensive as they would see an artificially high survival rate. Increasing the time between PN-collections beyond this inflection point can offer substantial performance improvements (see Section 3.1).

However, since PN-collection time is dominated by object processing and PN zeroing (and not things like stack walking overhead), if you extend the inter-PN-collection time further then the survival rate is relatively unchanged and thus little performance advantage is accrued. We came to this realization after an experiment we performed when we tried to reduce what seemed like an excessive number of PN-invariant collections in one of our benchmarks. We tried preventing these PN-invariant collections (and their presumed higher survivor rates) by adding private heaps as a third layer in the memory hierarchy. Like PNs, these private heaps would contain only a single thread's data and not be accessible by any other thread. (Unlike PNs, private heap size can expand on demand by commandeering as many empty public heap blocks as needed.) In this scenario, capacity PN-collections would move objects into a private heap instead of the public heap. If the write-barrier detected pointers to PN objects being stored in the private heap, rather than triggering a PN-collection, the write barrier stored the slot in a remembered set allocated using a reverse bump-pointer from the end of the PN. During the next PN-collection this remembered set was then enumerated as roots. If the write barrier detected a PN or private heap pointer being written into the public heap then a PN-invariant collection or the private heap equivalent was triggered, respectively.

For single-threaded applications, this approach typically reduced the number of PN-invariant collections to zero. While this did reduce the overhead from PN-collections, it did not significantly reduce the survivor rate and the additional overhead of maintaining the remembered sets resulted in no net gain in performance. Thus, despite the seemingly large number of PN collections that the application was causing, the time between PN-collections was still larger than the aforementioned inflection point. Thus, there was little reduction in the dominant object processing costs of the PN-collections.

5. Results

In this section, we present a performance comparison of TGC and JGC when both are paired with the rest of our experimental functional language system. In practice, TGC and JGC share much of the same code base including the management of public heap blocks, the distribution of public heap nurseries and the non-concurrent public heap collection algorithm. To this code base, TGC primarily adds private nurseries, the write barrier, and the concurrent collector. All tests were performed on a 4-socket Xeon with each socket containing a 6-core Intel Core 2 processor running at 2.1GHz for a total of 24 cores. Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 3MB L2 cache shared between two cores in addition to a common 16MB L3 cache. Unless otherwise stated, TGC results were obtained with a fixed-sized PN size of 128KB.

We present data for each of the six applications that have been written in our functional language. In Section 5.1 we examine our three parallel benchmarks: a raytracer, matrix multiply and Barnes-Hut. The raytracer was ported to our language from the PLClub OCaml winning entry to the 2000 ICFP programming contest. Matrix multiply was written from scratch. Barnes-Hut is an N-body physics simulation and was ported from SML. We also have three sequential benchmarks: cloth, sudoku, and SMVP. The cloth benchmark simulates the movement of a suspended piece of cloth given an initial impetus and was a port from C++. The sudoku benchmark is a sudoku puzzle solver and was written from scratch. Likewise, our SMVP benchmark implements a sparse matrix vector product and was written from scratch. The comparison between TGC and JGC for the sequential benchmarks is made in Section 5.2 along with a discussion of the performance effects of PN sizes. Section 5.3 shows results for TGC's concurrent collector where JGC is excluded as it has no concurrent mode.

5.1 Parallel Benchmark Results

In this section, we compare the execution times and scalability for our parallel benchmarks in non-concurrent mode. Figure 4 shows the execution times (all execution times in seconds) for our raytracing application rendering 10 complex 400x300 images. Here you can see that TGC provides a performance improvement over JGC even for small numbers of processors in which JGC does not saturate the bus. Figure 5 shows the corresponding scalability curves and illustrates that JGC's peak performance occurs around 8 processors whereas TGC's performance continues to improve up to 23 processors. Thus, much of the initial scalability limitations due to bus saturation that we saw for this application were indeed remedied by using TGC.

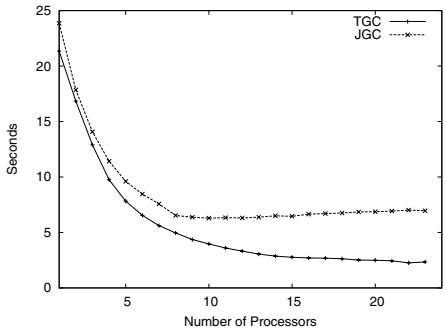

Figure 4. Raytracer Execution Times

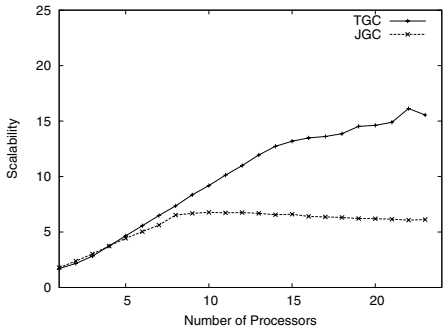

Figure 5. Raytracer Scalability

Figure 6 shows the execution times for our matrix multiply application. TGC consistently enables the application to run 22% faster than with JGC. Figure 7 shows the corresponding scalability curve for the matrix multiply application and indicates that both allow the application to scale nearly linearly.

Figure 8 shows the execution time for Barnes-Hut. Here again we see an across-the-board performance improvement of 25-50%

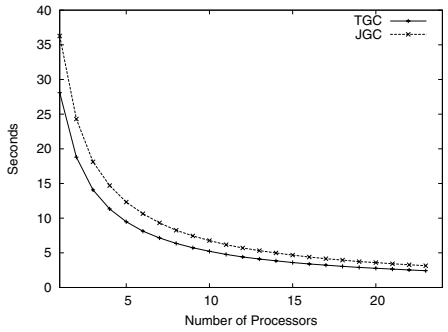

Figure 6. Matrix Multiply Execution Times

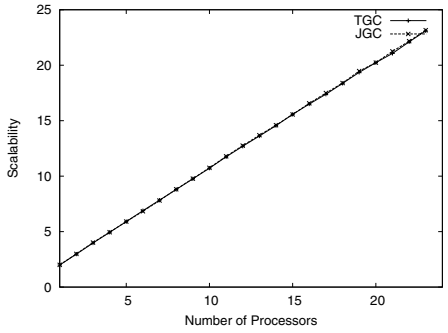

Figure 7. Matrix Multiply Scalability

for TGC while retaining the same scalability, Figure 9. In general, the scalability of Barnes-Hut was limited by the current benchmark having a large sequential component.

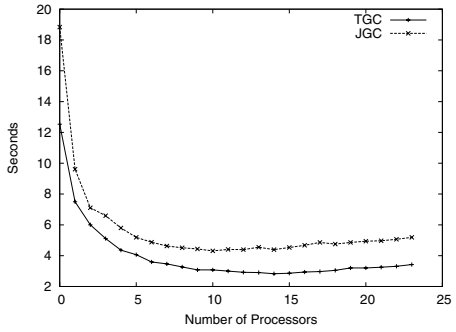

Figure 8. Barnes-Hut Execution Times

5.2 Private Nursery Sizes

This section presents data comparing TGC and JGC for our sequential benchmarks and discusses how the size of PNs effects performance. A variety of PN sizes are compared as well as the dynamic PN sizing technique (listed as "TGC Dynamic") described in Section 2.1.2. Figure 10 shows the execution times for the raytracer configuration as in Section 5.1. As can be seen, the raytracer is fairly insensitive to PN sizes. Our results also indicate that execution times increase steadily if you decrease PN sizes below 64K. Likewise, the results for matrix-multiply indicate that it is similar to raytracer in its relative insensitivity to PN sizes.

Figure 11 shows the execution times for our sudoku benchmark under JGC and TGC with PN sizes ranging from 4MB to 256KB as

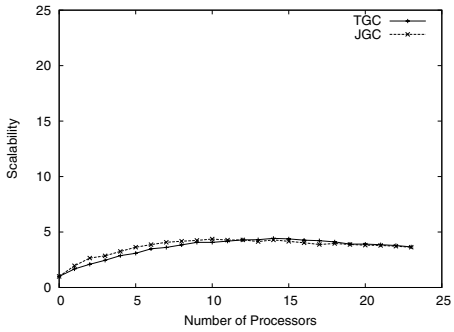

Figure 9. Barnes-Hut Scalability

Configuration	1 proc	8 procs	16 procs	24 procs
TGC PN=4MB	36.12	5.50	3.09	2.33
TGC PN=2MB	36.19	5.52	3.09	2.36
TGC PN=1MB	36.08	5.52	3.06	2.36
TGC PN=512KB	36.15	5.52	3.09	2.34
TGC PN=256KB	36.26	5.53	3.07	2.34
TGC PN=128KB	36.31	5.53	3.09	2.34
TGC PN=64KB	37.42	5.51	3.06	2.34
TGC Dynamic	36.17	5.53	3.06	2.34

Figure 10. Raytracer Execution Times for Various PN Sizes

Configuration	Sudoku	Relative Increase to Best
JGC	84.66	12.8%
TGC PN=4MB	90.63	20.76%
TGC PN=2MB	75.05	0.0%
TGC PN=1MB	76.35	1.73%
TGC PN=512KB	77.95	3.87%
TGC PN=256KB	80.00	6.60%
TGC Dynamic	75.11	0.08%

Figure 11. Sudoku Execution Times for Various PN Sizes

well as TGC in PN-dynamic-sizing mode. In this case, 2MB was the best fixed PN size performing 13% better than JGC and the dynamic sizing algorithm performed only 0.08% worse. Here, the dynamic sizing algorithm terminated with 2.09MB as the PN size.

Figure 12 shows the execution times for our cloth benchmark, again comparing JGC with various PN sizes and the dynamic scheme. For this benchmark, only a PN size of 9MB was faster than JGC (by 5.4%). In this case, the dynamic scheme did not perform as well with a slowdown of 27% compared to optimal. The dynamic scheme selected a PN size of 8.39MB. When the benchmark was run with a fixed PN size equal to 8.39MB, the result was 10.61s. This time is a 15% slowdown compared to the best fixed PN size observed and indicates that the dynamic scheme itself is adding an additional 12% in overhead. We hope to reduce both of these numbers in the future.

The results from Figure 12 are fairly regular, decreasing from 16MB down to 9MB and then increasing again as the PN size is reduced further to 4MB. The exception to this regularity is the point at 6MB. The phenomenon seen at that point is one in which a fortunate alignment of application characteristics and PN size causes the write barrier location having the lowest associated PN-collection survival rate to increasingly become the predominant PN-collection triggering point. The PN-collection survival rate at 6MB was 4.7% whereas the rates for 5MB and 7MB were 7.1%

Configuration	Cloth	Relative Increase to Best
JGC	9.69	5.4%
TGC PN=16MB	13.48	46.7%
TGC PN=15MB	11.39	23.9%
TGC PN=14MB	12.13	32.0%
TGC PN=13MB	12.1	31.7%
TGC PN=12MB	10.4	13.2%
TGC PN=11MB	11.2	21.9%
TGC PN=10MB	10.7	16.4%
TGC PN=9MB	9.19	0.0%
TGC PN=8MB	11.58	26.0%
TGC PN=7MB	13.78	49.9%
TGC PN=6MB	10.72	16.6%
TGC PN=5MB	14.55	58.3%
TGC PN=4MB	16.44	78.9%
TGC Dynamic	11.7	27.3%

Figure 12. Cloth Execution Times for Various PN Sizes

Configuration	SMVP	Relative Increase to Best
JGC	55.49	74.3%
TGC PN=4MB	35.88	12.7%
TGC PN=2MB	37.16	16.7%
TGC PN=1MB	32.24	1.3%
TGC PN=512K	32.03	0.6%
TGC PN=256K	32.00	0.5%
TGC PN=128K	31.92	0.3%
TGC PN=64K	31.83	0.0%
TGC PN=32K	31.97	0.4%
TGC Dynamic	32.03	0.64%

Figure 13. SMVP Execution Times for Various PN Sizes

and 6.0% respectively. To a somewhat lesser extent we observe this phenomenon at the 9MB point as well.

Figure 13 shows the execution times for our SMVP benchmark. For this application, a substantial improvement is seen for all TGC configurations when compared to JGC. The sensitivity to PN size below 2MB is low and the dynamic scheme has less than 1% slowdown over optimum. In this case, the dynamic scheme selected 524K as the PN size.

5.3 Concurrent Mode Comparisons

In this section, we compare TGC against itself in each of its three operating modes: non-concurrent, concurrent, and concurrent minimal (Section 2.2.1). Figure 14 presents this comparison for the raytracer. The concurrent GC in minimal mode is 11% slower than non-concurrent mode at two processors and averages 10% slower for 8 or more processors. Conversely, non-minimal concurrent GC mode averages 17% slower for 8 or more processors. For 24 processors, the average thread pause time for PN-collection or major GC was 4us. The maximum pause time for the steady state (after startup) portion of the raytracer was 5ms.

Figure 15 shows the same 3-way comparison but for the matrix multiply application. Here again, minimal mode reduced the number of GCs performed from 142 in continual mode (26.581s of total concurrent GC time) to 36 with minimal mode (16.097s of total concurrent GC time) at 16 processors. Thus, the per-GC cost is 2.4 times higher in minimal mode than in continual mode due largely in part to larger number of blocks that need to be swept per GC in minimal mode because the larger inter-GC time allows more blocks to be filled (the continual collector encounters more blocks known to be completely free of data and thus not in need of being swept).

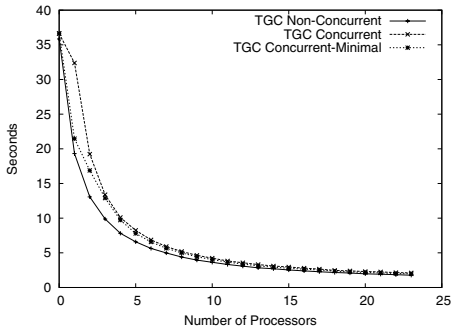

Figure 14. Raytracer Execution Times - Comparing Non-Concurrent vs. Concurrent

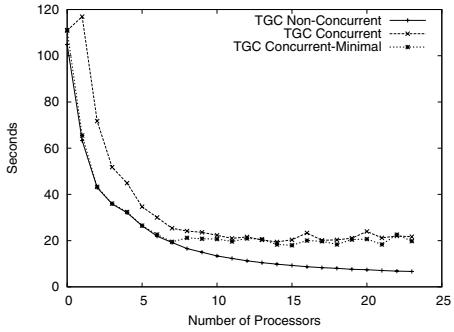

Figure 15. Matrix Multiply Execution Times Comparing Non-Concurrent vs. Concurrent

6. Related Work

Parts of the nursery ping-ponging effect are addressed in Sun's Java HotSpot GC with the +UseNUMA option [2]. This option breaks up the younger generation into chunks which are then fixed on individual processors. Thus, the issue of a nurseries migrating between processors as the thread migrated is eliminated. However, in the HotSpot GC, the collection of those nurseries would likely take place on a different processor.

Doligez-Leroy-Gonthier [7, 8] describe a concurrent GC for ML in which immutable objects are allocated in a thread-local private heap, which itself is collected when full while the thread is stopped. Public heap to private heap pointers are prevented using a write barrier which copies the transitive closure of reachable objects from the object escaping the private heap into the public heap. The public heap is collected concurrently with the mutator threads. The Manticore project [11] uses a combination of the DLG and Appel [3] collectors.

Domani, et al. [10] present a generational, concurrent collector for Java in which both the old and young generations are collected concurrently, proceeding along the lines of the DLG collector. Unlike DLG, this collector does not use private heaps as Java lacks sufficient mutability information and the cost of moving mutable objects concurrently is too high. However, in [9], Domani, et al. do describe a non-concurrent Java GC that uses "thread-local heaps." These heaps are thread-local only from the perspective of allocation. A write barrier is used to dynamically determine when an object becomes public. This write barrier uses an external bitmap which records which objects are global and which are thread-local.

If the write barrier detects a public-to-private write, the barrier updates this bitmap to reflect that the private object is now public. Thus, globally accessible objects can exist in any thread-local heap and be accessed by any thread. Collections of the thread-local heaps are mark-sweep and create free lists. Small objects are allocated via a bump pointer from some thread-local heap free-list entry. Medium sized objects are allocated from the thread-local heap free-list by first fit. Large objects are allocated from a global pool. Extensions to their base algorithm are also presented that allow compaction of global and local objects.

Steensgaard describes a collector [16] with a public heap and thread-local heaps. Static escape analysis is used to determine which objects may escape the thread and those are initially allocated in the public heap. All heaps are generational, the younger generation collected by copying into the older and the older using two-space copying techniques. Two kinds of garbage collection were allowed: 1) the collection of all of the young generations of all the heaps (including the public heap) and 2) the collection of all heaps (public and private, young and old). Individual collections of thread-local heaps was not possible.

Several works have explored the technique of overwriting the return address or otherwise trapping a returning function to facilitate stack scanning. Yuasa et al. [19] describe a way to overwrite the return address of a frame with a "return barrier" in an incremental stack scanning scheme to mark the boundary between the scanned and unscanned portion of the stack. If a mutator tries to return across this barrier, the barrier triggers some additional stack scanning and then allows the mutator to continue. A similar return barrier is used in [4] to detect whether a stack has been modified since the last GC epoch to prevent redundant work. In [15], Kliot et al. describe a mechanism which uses a similar return barrier but which allows a collector thread and a mutator to cooperate in the incremental scanning of a stack using lock-free synchronization. Cheng et al. [5] describe another method for limiting stack walking of deeply recursive functional language stacks using a return barrier on every 25th frame and maintaining bookkeeping information for previously seen frames with special handling for exception processing. In contrast, our watermarking scheme does not require any corresponding bookkeeping information and so we are able to be more accurate and watermark every frame previously seen during a stackwalk.

Likewise, several works have proposed other mechanisms to dynamically resize the younger generation [3, 12, 17, 18].

7. Conclusions

We have presented a GC designed to minimize bus traffic by the use of permanent, private, thread-local nurseries. These nurseries are collected independently of one another and those collections reclaim up to 99% of all allocated objects. We use a write barrier to maintain the invariant that a public object may not point to a private one and use mutability information to optimize this write barrier to duplicate immutable objects in the public space that would otherwise require a PN-collection. Nevertheless, the relatively small PN sizes leads to many PN-collections which puts pressure on parts of the system such as stackwalking. As such, we presented a frame watermarking mechanism that avoids reprocessing frames during PN-collections. We presented a scheme for dynamic optimization of the PN size and discuss potential optimizations for the timing of PN-collections. We described three major GC implementations including a scheme to run the concurrent collector only as much as necessary to keep up with the public heap block allocation rate. The concurrent minimal major GC mode averages 16% less throughput compared to the stop-the-world major GC mode but maintains average pause times less than 10us and maximum pause times less than 5ms. Finally, we again make use of mutability information to allow the compaction of immutable public heap objects while avoiding a stop-the-world phase or any atomic pointer flipping and see reductions of fragmentation up to 99%. We saw single-threaded performance improvements across a range of benchmarks from 5-74% and sufficient reduction in bus traffic to allow one previously memory-bound parallel benchmark to increase scalability by 10x.

References

[1] Intel VTune Performance Analyzer. Online `http://www.intel.com/software/products/vtune/index.htm`.

[2] Java HotSpot Virtual Machine Performance Enhancements JDK 7. Online `http://download.java.net/jdk7/docs/technotes/guides/vm/performance-enhancements-7.html`.

[3] A. W. Appel. Simple generational garbage collection and fast allocation, 1988.

[4] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (Snowbird*, pages 92–103. ACM Press, 2001.

[5] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *PLDI*, pages 162–173, 1998.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. In *Communications of the ACM*, pages 966–975, 1978.

[7] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, pages 70–83, 1994.

[8] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL*, pages 113–123, 1993.

[9] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 76–87, New York, NY, USA, 2002. ACM.

[10] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for Java. *SIGPLAN Not.*, 35(5):274–284, 2000.

[11] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: the manticore project. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, New York, NY, USA, 2007.

[12] X. Guan, W. Srisa-an, and C. Jia. Investigating the effects of using different nursery sizing policies on performance. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 59–68, New York, NY, USA, 2009. ACM.

[13] R. Hudson, J. Moss, A. Diwan, and C. Weight. A language-independent garbage collector toolkit. Technical Report UM-CS-1991-047, University of Massachusetts, 1991.

[14] R. Hudson, J. Moss, S. Subramoney, and W. Washburn. Cycles to recycle: garbage collection to the ia-64. In *ISMM 2000*, 2000.

[15] G. Kliot, E. Petrank, and B. Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20, New York, NY, USA, 2009. ACM.

[16] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM*, pages 18–24, 2000.

[17] J. M. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic management of nursery space organization in generational collection. *Interaction between Compilers and Computer Architecture, Annual Workshop on*, 0:33–40, 2004.

[18] T. Yang, E. Berger, M. Hertz, S. Kaplan, and J. Moss. Automatic heap sizing: Taking real memory into account, 2004.

[19] T. Yuasa, Y. Nakagawa, T. Komiya, and M. Yasugi. Return barrier. In *International Lisp Conference*, 2002.