Task-Aware Garbage Collection in a Multi-Tasking Virtual Machine

Sunil Soman
Computer Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
sunils@cs.ucsb.edu

Laurent Daynès
Sun Microsystems Inc.
180 Av. de L'Europe
ZIRST de Montbonnot
Montbonnot St-Martin 38330, France
laurent.daynes@sun.com

Chandra Krintz
Computer Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
ckrintz@cs.ucsb.edu

Abstract

A multi-tasking virtual machine (MVM) executes multiple programs in isolation, within a single operating system process. The goal of a MVM is to improve startup time, overall system throughput, and performance, by effective reuse and sharing of system resources across programs (tasks). However, multitasking also mandates a memory management system capable of offering a guarantee of isolation with respect to garbage collection costs, accounting of memory usage, and timely reclamation of heap resources upon task termination.

To this end, we investigate and evaluate, novel *task-aware* extensions to a state-of-the-art MVM garbage collector (GC). Our task-aware GC exploits the generational garbage collection hypothesis, in the context of multiple tasks, to provide performance isolation by maintaining task-private young generations. Task aware GC facilitates concurrent per-task allocation and promotion, and minimizes synchronization and scanning overhead. In addition, we efficiently track per-task heap usage to enable GC-free reclamation upon task termination. Moreover, we couple these techniques with a light-weight synchronization mechanism that enables per-task minor collection, concurrently with allocation by other tasks.

We empirically evaluate the efficiency, scalability, and throughput that our task-aware GC system enables.

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

General Terms Design, Performance, Experimentation, Algorithms

Keywords Task-aware garbage collection, multi-tasking, resource reclamation, virtual machine, Java

1. Introduction

Managed runtime environments (MREs) for modern object-oriented programming languages, enable portability and mobility through the use of an architecture-independent program transfer format. MREs, such as the Java^{TM} Virtual Machine (JVM), and the Microsoft .NET framework, typically load, compile, and optimize programs incrementally and dynamically for efficient execution on the underlying hardware platform. MREs commonly execute a single program with a single MRE instance, and rely on the underlying operating system to isolate programs from each other for security, as well as for resource management and accounting.

Unfortunately program isolation at the granularity of the virtual machine can significantly restrict the performance of MREs that execute multiple, independent, programs concurrently. This execution model duplicates effort across VM instances, since it prohibits sharing of MRE services and internal representations, memory, code, etc., across programs. Such redundancy increases startup time and memory consumption and degrades overall system performance and scalability.

A multitasking implementation of a MRE, such as the multitasking virtual machine (MVM) can address these problems while maintaining portability, mobility, and type-safety. MVM executes multiple programs within a single operating system process using isolating units called *tasks* [8]. Co-locating programs in the same address space simplifies the virtual machine implementation through sharing of the runtime representation of programs and dynamically compiled code. Such sharing also avoids duplicated effort across programs (e.g. loading, verification) and amortizes runtime costs, such as dynamic compilation, over multiple program instances. Prior work on the MVM [7], shows how a multitasking design reduces startup time and memory footprint, and improves performance over a single-program MRE approach.

The focus of our work is on memory management for multitasking MRE, and MVM in particular. An MVM GC implementation must address unique challenges not faced by GC systems within single-tasking MREs to achieve scalable performance.

First, each program (task) that executes must not interfere with other programs, either functionally, or in terms of performance. In particular, a GC that one task triggers should not impact the performance of other tasks. Moreover, GC overhead for a task should be independent of the number of tasks executing. In addition, the task should have control over heap and GC system parameters, such sizing or generational tenuring parameters. Finally, upon task termination, heap resources that the task has allocated must be immediately reclaimable and available for use by other tasks. In addition, such reclamation should not adversely affect other tasks.

The current MVM system [1] implements a simple memory management system in which a single heap and management policy is shared across all tasks. Such sharing allows tasks to interfere with one another (in terms of performance), and restricts the scalability of the system. Moreover, there is no per-task control over GC parameters or reclamation of heap resources upon task termination

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
ACM 1-59593-221-6/06/0006.

without requiring an expensive, full heap GC. Extant multi-tasking approaches (e.g. [6]), that do not employ MRE support impose similar restrictions. An alternative approach is to assign a separate heap space (and possibly different GC policies) to each task. Using such an approach complicates the memory management system, restricts the the opportunistic use of reserved idle memory by other tasks, and can limit the number of concurrent tasks that the system can support.

In this paper, we present a design that addresses these challenges for the Sun Microsystems MVM [1] for the Java^{TM} programming language. Key to our design is an organization of the heap that enables (i) per-task performance isolation for the memory management system, (ii) independent allocation and collection of young objects, and (iii) GC-free memory reclamation upon task termination.

The design follows a hybrid approach that divides the heap into task-private and shared sections, so that we confine a majority of GC activity to task-private sections. This hybrid organization of the heap works particularly well with generational GC algorithms that divide the heap into multiple generations. Generational GCs segregate objects by age and concentrate their GC efforts on the youngest generations (i.e., the generations holding the youngest objects) by exploiting the weak generational hypothesis [29], which states that most objects die young.

In our implementation, the heap consists of multiple *independent young generations* (one per running task), and a single old generation that all tasks share. When a task enters the system, it is given a private young generation that the system sizes according to parameters specified by the task. A task allocates primarily from its young generation. When this area is full, the system performs a *minor collection for that task*. During a minor collection, the GC system moves (promotes) mature live objects to the shared old generation.

The shared old generation efficiently tracks the regions that each task consumes using *promotion allocation buffers* (PABs). A PAB is a region of old generation space that the system assigns to a particular task for allocation. A task uses its PAB for both object promotion during minor collections and for direct (pretenuring) allocation of objects. PABs provide numerous advantages – they cluster objects of the same task together in the shared old generation, they ease accounting of space consumed by tasks in the shared generation, they enable immediate reclamation of old generation space without garbage collection upon task termination, and they help limit the amount of old generation space that the system must scan to identify roots during minor collection. In addition, by combining per-task young generations with PABs, we eliminate interference between mutators and collectors of different tasks. As a result, our system is able to perform minor collection for a task, concurrently with the execution of mutators of other tasks.

In summary, we contribute the following,

- A multitasking-aware garbage collected heap design that improves prior work by supporting instantaneous, collection-less reclamation of all heap space of terminated tasks, minor collection of one task concurrently with mutation by other tasks, and a method for identifying roots of minor collection that is independent of the number of tasks,
- An empirical evaluation of the impact on performance and scalability of task-aware GC techniques (independent young generation, promotion allocation buffers, concurrent minor collection and mutation), using multitasking workloads derived from a variety of programs.

The rest of this paper is organized as follows. Section 2 provides background on the MVM, a Sun Laboratories' implementation of a multitasking virtual machine based on the Java HotSpot^{TM} virtual machine version 1.5 [27]. The current prototype of MVM [1], which we employ and extend this MVM in this paper, implements a naïve approach towards heap management, since all tasks share the same generational heap. Section 3 details our hybrid approach for task-aware garbage collection. Section 4 reports the results of the experiments we have conducted to evaluate the impact of the mechanisms added to MVM – per-task young generation, PABs, and concurrent minor collection and mutation. Section 5 discusses related work, and Section 6 summarizes our findings.

2. Multi-Tasking Virtual Machine (MVM)

MVM [7] is an implementation of the JVM that co-locates execution of multiple programs in a single operating system process. Each program execution is carried out as a *task*. Tasks are used to implement *isolate*s, which are execution containers for arbitrary programs formally defined by the Application Isolation API (Java Specification Request 121) [19].

Isolates provide a program with the illusion of a stand-alone JVM. Programs have the same behavior as if they were running on a private JVM. Each isolate has its own primordial loader and hierarchy of class loaders. No sharing of objects can take place between isolates, and the JVM safeguards against inter-isolate interference.

Each task in MVM is associated with a unique task identifier. A task identifier is an index into tables used in MVM to mediate access to data structures that need to be replicated on a per-task basis, such as, the task specific part of the runtime representation of a class. All threads running in the context of a given task, are associated with the identifier of that task as well as other relevant task-specific information. We next describe the MVM features that are pertinent to memory management.

2.1 Class Sharing

MVM substantially reduces the footprint of programs by implementing a form of sharing of the runtime representation of classes called *task re-entrance* [10]. Task re-entrance is supported only for classes defined by class loaders, whose behavior is fully controlled by the MVM. This includes the *primordial* and *system* loader of each isolate.

The primordial loader is a special class loader that bootstraps class loading. It is used to load the *base* classes that are intimately associated with an implementation of the JVM and are essential to its functioning (such as classes of the `java.*` packages). The system loader is the loader that defines the main class of a program. It typically obtains class files from the local file system at a fixed location specified at program start-up.

The system loader serves class loading requests by first delegating them to the primordial loader, and only defines classes that the primordial loader does not define. This behavior is predictable since for a given class path, a class loaded by a primordial or a system loader of any task is always built from the same class file. Further, symbolic references from classes defined by a primordial or a system loader always resolve identically across tasks.

This allows for a simplified form of sharing where only the task-dependent parts of the runtime representation of a class, such as static variables, class initialization state, protection domain, instance of `java.lang.Class` etc., need to be replicated per loader. All other class information, in particular those derived from resolved symbolic links, such as field offsets, virtual table indexes, static method addresses, etc., can be shared across loaders, further increasing the amount of sharing. Access to the task-private part of the representation of a class shared across multiple tasks is mediated via a table indexed by a task identifier (task id). Sharing is not supported for classes defined by program-defined loaders. Instead of a table of task-private class representations, the class representation includes a single task-private representation. Both the inter-

preter and code produced by the dynamic compiler are aware of this organization and access the task-dependent class information using the task identifier of the current thread.

An extensive description of how MVM implements sharing of the runtime representation of classes, including bytecode and code produced by the dynamic compiler, is described in [7].

2.2 Garbage Collection

The MVM derives from the HotSpot^{TM} Java virtual machine [23]. The current prototype of the MVM [1] retains the heap layout of the original Java HotSpot virtual machine and introduces minor changes. Heap management follows a generational strategy based on three generations – permanent, tenured, and young. The permanent generation is a special generation used for allocating objects that constitute the runtime representation of classes and string literals. In the MVM, the permanent generation also includes task tables associated with the runtime representation of task-reentrant classes. Note, however, that we do not allocate the task-private representation of a task re-entrant class, which holds static variables etc., in the permanent generation but, rather, in the tenured generation. The rationale for this is that in the MVM, the lifetime of the sharable part of the runtime representation of a class is much longer. The sharable part's lifetime may range from the lifetimes of a few tasks to the lifetime of the virtual machine itself, unlike the task-private part, which lasts no longer than the duration of the task. The task-private part of the runtime representation of classes is allocated directly in the tenured generation. This avoids cluttering the young generation with objects known to be long lived.

Program threads allocate from the young generation. As in the original Java HotSpot virtual machine, the young generation is divided into an allocation space (the *eden*), and a mature space [1], which consists of a pair of equally sized semi-spaces (a *from* and *to* space). Garbage collection of the young generation uses a copying scavenger that evacuates live objects from the *eden* and *from* spaces to the *to* space according to a design similar to that in [28]. Mature objects that have survived several scavenge cycles are promoted to the old generation. Objects from the young generation are never promoted to the permanent generation.

The eden space is used for the vast majority of allocations. Objects that do not fit in the young generation are allocated directly in the tenured generation. To increase per-thread locality and to avoid the cost of atomic instructions in allocation code, the system allocates a thread local allocation buffer (TLAB) from the eden space for threads of tasks. Write barriers for tracking cross-generation pointers follow a card-marking scheme.

The Java HotSpot virtual machine supports several algorithms for the tenured generation, but MVM currently only supports mark and compact. Both minor and major collections require bringing all threads to a safepoint in order to proceed. In the case of MVM, all threads of all tasks must be at a safepoint.

The changes introduced by MVM to garbage collection are related to reclaiming space used by terminated tasks. MVM maintains a list of terminated tasks which is purged on a garbage collection. During collection, the list of terminated tasks is used to scan task tables of the runtime representation of classes, and other task tables referring to heap objects, to zero-out the entries corresponding to terminated tasks, so that no objects of the terminated task are reachable from any live root. This clean up is performed at garbage collection time rather than at task termination, since (i) the heap space used by terminated task cannot be reclaimed without performing a full GC, and (ii) postponing clean up until GC enables the system to factor out the cost of clearing dead references from entries of task tables corresponding to terminated tasks.

[1] Should not be confused with the old generation

3. Task-Aware Garbage Collection

Although simple, the approach of sharing a generational heap between a dynamically varying number of independent tasks presents several problems. First is the absence of *performance isolation* with respect to garbage collection. That is, garbage collection affects all tasks at once, and has a cost proportional to the live objects of all tasks. A second problem is the inability to immediately reclaim the heap space consumed by a task, upon its termination. Resource reclamation requires a full garbage collection, which affects all tasks. Both problems adversely impact scalability and response time.

The following section presents a generational garbage collection system that attempts to better address the requirements of MVM with a combination of three features – per-task independent young generations, promotion allocation buffers, and and task-concurrent scavenging.

3.1 Hybrid generational heap

The first element of the design builds on [7], by providing each task with a private young generation, while sharing a single old generation between all tasks. This hybrid approach attempts a compromise between sharing the heap between all tasks and giving each task an independent heap. There are severals reasons for this choice.

First, the young generation is typically much smaller than the old generation. Thus, having one per task young generation and sharing the tenured space makes better use of heap resources, by avoiding committing too much memory per task, and unnecessarily limiting the degree of multitasking. Old generation space is allocated to a task on demand, either during minor collection, or when pre-tenuring objects.

Second, the vast majority of allocations and most garbage collections occur over the young generation. Thus, an independent young generation shields a task from most heap-related interference, especially varying allocation rate, tenuring decisions, and interleaving of objects from different tasks. Also, minor collection pauses are proportional to the live set of objects of a given task, as opposed to all tasks.

Third, key parameters for generational garbage collection, such as young generation size, age-based tenuring policy, etc., can be controlled on a per-task basis. This enables users to specify an appropriate application-specific set of tuning parameters for each task.

Figure 1 depicts our layout for per-task young generations. A table, called the young generation virtualizer, maps the task identifier to the corresponding young generation. Each young generation has the same layout as in the original Java HotSpot virtual machine (see section 2), with the notable exception that a young generation may consist of several discontinuous regions of memory. Specifically, space for young generations is allocated from a pool of fixed-sized chunks, the size of which is parameterizable and set at 2MB by default. On startup, a task is allocated an integral number of chunks corresponding to the size of the young generation requested (or the default if none is specified). The chunk manager attempts to allocate contiguous chunks when possible, otherwise, it assigns additional edens to the young generation, one per region of contiguous chunks allocated from the pool (similarly to the surplus memory in [7]). The pool manager may re-arrange the chunks allocated to a young generation to reduce the fragmentation of its eden. Such rearrangement takes place as necessary following minor collection, when all the live objects of the eden space have been evacuated. This organization also allows us to dynamically change the size of a young generation at runtime.

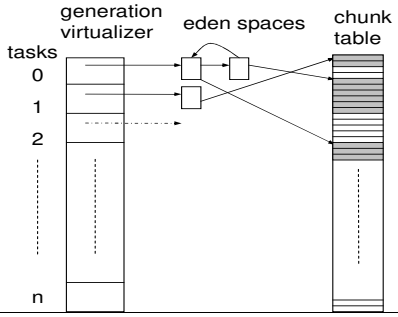

Figure 1. Task independent flexible young generations. A generation virtualizer maps tasks to young generations. Each generation comprises one or more eden spaces, each of which consists of an integral number of contiguous chunks allocated from a pool. Eden spaces of a task are linked together. Chunks can be added or removed dynamically.

The to and from spaces are typically much smaller than the eden space. For simplicity, the current prototype limits their size to that of a single chunk.

As in the original Java HotSpot virtual machine, threads are assigned one or more thread local allocation buffers (TLABs) so that they can allocate objects without synchronization with other threads. The TLAB of a thread is allocated from the eden of the young generation of the thread's task.

Per task young generations provide some degree of performance isolation – the copying cost of scavenging is proportional to the number of live objects of the task that triggered the scavenge; further, only mature objects of that task are promoted to the shared old generation.

However, per task young generations alone are insufficient for complete performance isolation. All tasks must still be stopped at a safepoint in order for the scavenger to have a consistent view of the old generation. In particular, consistency of the remembered set of references to young generations must be guaranteed, in order to precisely locate references from the old generation, to the young generation being scavenged. Note, however, that tasks are stopped at the safepoint only for the duration of the scavenge of the live objects of a single task, which improves over a design that shares a single young generation between tasks.

Another concern is that per task young generations do not enable immediate reclamation of all heap space consumed by a terminated task. The young generation can only be reclaimed when there are no more no references to it from the old generation. Otherwise, it may lead to situations where an obsolete pointer from the old generation may be mistaken for a valid pointer if the reclaimed space has been re-allocated for the young generation of another task. For this reason, young generation space can only be freed once all such references have been cleared. This can be done opportunistically at any scavenge, while scanning the remembered set. In addition, space consumed by a terminated task in the old generation can only be reclaimed upon a full collection of the old generation.

To address the problems listed above, we complement per task young generations with promotion allocation buffers (PABs). PABs allow instantaneous, collection-less, reclamation of all heap space (i.e., both young and old) consumed by a terminated task. PABs also help to simplify synchronization issues towards efficient support for task-concurrent scavenging.

3.2 Promotion Allocation Buffers

Immediate, collection-less reclaiming of the heap space used by a terminated task can be obtained by precisely tracking old generation regions in which objects allocated by each task reside. With this knowledge, young generation collection can ignore all regions of the old generation that do not contain objects of the task being scavenged, since these are not required to determine roots for collection. Since no regions of old generation that may contain obsolete references to young generations of terminated tasks will be scanned, young generations of terminated tasks can be re-used immediately, without GC.

The old generation space used by a terminated task can be reused immediately without any collection as well. The only references to regions used by a terminated task originate from the tables used to mediate access from the shared part of the runtime representation of classes stored in the permanent generation, to their task-private parts located in the old generation. Thus, the regions corresponding to a terminated task can be immediately re-used, if the GC ignores entries of the tables corresponding to terminated tasks. This, however, prevents re-use of the identifiers of terminated tasks. These identifiers will eventually need to be reclaimed by cleaning corresponding entries in the global task table. The cleaning of these entries can be done opportunistically on the next GC that requires scanning the task table, or by a separate background thread. Note that cleaning itself does not require any synchronization with tasks.

Precisely identifying which regions of the shared old generation hold objects of a terminated task is key to the collection-less reclamation of the heap space used by the terminated task as described above. Tracking individual objects would likely be prohibitively expensive. Instead, we propose promotion allocation buffers (PABs).

A PAB is a contiguous region of the old generation assigned to a task. PABs are primarily used during scavenging of the young generation of task when promoting young objects to the old generation. They are also used for the occasional direct allocation of objects in the old generation, either because the object does not fit in the young generation, or as a result of a pre-tenuring decision. For example, as described previously, the task-private representation of a class is always pre-tenured. The size of a PAB can be task-specific and adjusted dynamically. It is generally chosen to satisfy several scavenges. Allocation in a PAB involves increasing a cursor to the first free byte in the PAB (bump-pointer). When allocation in a PAB is performed by mutator threads, synchronization between threads is required, since the PAB of a task is shared between all threads of the task.

Figure 2 illustrates PAB management. Each task is associated with a current PAB and a list of full PABs. An initial PAB is allocated to a task at startup, prior to the first allocation by the task. When a PAB is full (typically during a scavenge), its address is recorded in the task's list of full PABs, and a new one is provided to the task.

If an object does not fit in a PAB, space is allocated directly from the old generation, either from a previously freed PAB, or from the free space at the end of the old generation (beyond the last PAB). In both cases, the object is recorded as a full PABs in a list corresponding to the task performing allocation. The list of full PABs, thus, precisely tracks regions of the old generation used by a task.

When a task completes, its task identifier is added to a list of tasks whose task table entries can be freed and re-used. The task's current PAB and full PABs are added to a global list of free PABs, and become immediately available for re-use by other tasks. Young generation chunks of the task are returned to the global pool, and are immediately available for re-use by the young generations of other tasks (see Section 3.1).

Adjacent free PABs are coalesced in a single PAB. Free PABs at the end of the old generation are removed from the list and the pointer to top of the old generation is updated accordingly, as illustrated in Figure 3. Apart from limiting the space overhead

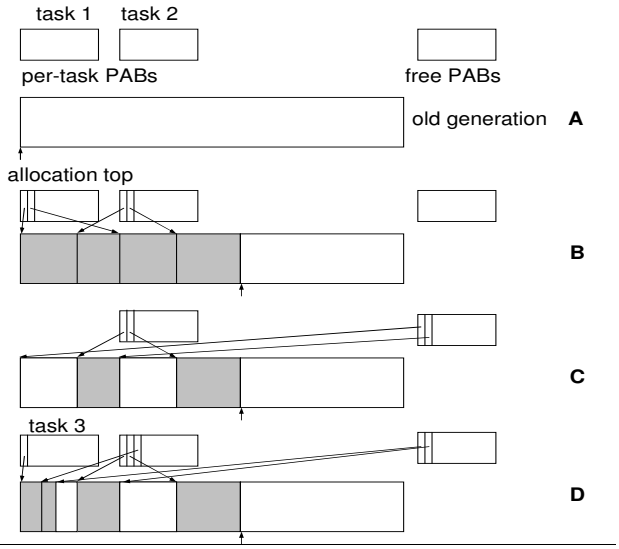

Figure 2. Example of PAB management & tenured space reclamation at task termination without a full GC. (A) Initial configuration. (B) Both tasks 1 & 2 have performed promotions and their respective full PAB list are now non-empty. (C) Task 1 terminates and its set of full PABs is added to the global free list. (D) Task 3 enters the system and task 2 & 3 start using space allocated from the PAB free list.

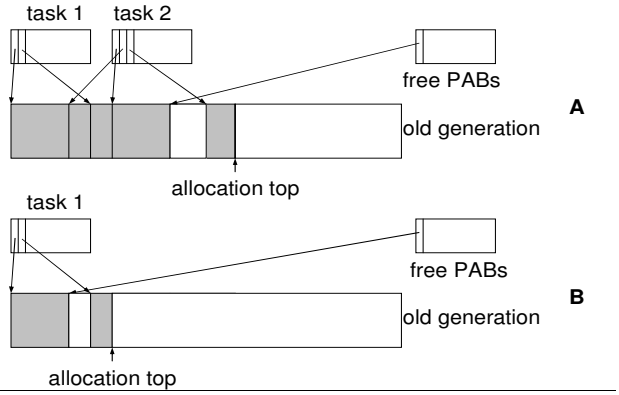

Figure 3. Example illustrating shrinking of old generation footprint upon task termination.

of tracking PABs, coalescing can increase the size of contiguous free PAB areas, consequently limiting fragmentation and further reducing the frequency of full GC.

PABs have several interesting properties. First, they improve isolation between tasks, since most allocation in the shared old generation is performed from a PAB that is private to one task, eliminating a point of interference between tasks. Second, they efficiently keep track of the old generation space used by tasks. Tracking is relatively inexpensive and only involves adding a PAB to a list of full PABs when a PAB is full, or when an object larger than the current PAB capacity is allocated. This precise tracking enables collection-less reclamation of both young and old generations space used by terminated tasks. Further, it optimizes the identification of references from the old generation to a particular young generation (as will be described later). Last, it enables precise accounting of space consumed by tasks.

Maintaining PABs Across Major GCs

Reclamation and reuse of PABs mitigates full heap GC, but is not a replacement for it. The old generation may fill up eventually, requiring collection. A sliding mark-compact collector is used for the old generation. The collector may reclaim garbage in PABs and compact live objects inside PABs, thus invalidating their original boundaries. Consequently, old generation collections may require adjustment to PABs boundaries. The following describes how this adjustment is performed (Figure 4).

The old generation mark-compact GC is a standard 4 phase compacting collector [22] involving the following phases.

- Mark live objects.
- Compute new addresses for live objects.
- Scan objects and adjust references to point to the new locations.
- Relocate (copy) objects to their new locations.

Adjustment of PAB boundaries can be performed between the second and third phases. During the second phase (computing new addresses), the GC stores the new address for a live object that will be relocated, in the object header. To compute new boundaries for a particular PAB, we locate the address of the first live object in the PAB. If no live object is found, this PAB can be dropped from the corresponding task's list. If a live object is found, we read its new address from the header, which now becomes the new start of the PAB. To adjust the end of a PAB, we note that the first live object beyond the end of the PAB would be moved to a location right after the new end of the PAB. The new end of the PAB is therefore the new location of the first live object past the current end.

Note that locating the first live object from either the start or end of a PAB can be expensive. However, we make use of an optimization that the existing garbage collector itself uses to quickly skip over dead objects. During the second phase of mark-compaction, the GC records the address of the next live object in the header of the first dead object in a group of contiguous dead objects. In the best case, the current boundary of a promotion area is the first dead word in a group of dead objects. However, this may not always be the case, hence, we may need to iterate over successive dead objects until we find the next live (GC marked) object. To avoid excessive scanning, it may be necessary to limit the number of dead objects scanned, and discard the PAB entirely if this number is over a threshold. In practice, we find that this overhead is not excessive. Note that discarding PABs does not affect correctness.

Optimizing Scavenging

Scavenging uses a card table [5, 17, 15] to identify references from the shared old generation to per-task young generations, in order to identify reachable young objects. In the presence of a large number of dirty cards belonging to different tasks, scanning the entire set of dirty cards at each scavenge might prove expensive. The existing card table implementation does not associate cards with tasks and hence, every scavenge requires scanning all dirty cards. Having mutators record task information in cards would add an additional cost to the write barrier, thus negating an important advantage of using a card table. In addition, extra space per card would be needed to record a task identifier, or a list of task identifiers.

Our scheme of tracking per-task old generation usage via PABs can be readily used to scan dirty cards of only the task initiating the scavenge. This substantially reduces the number of cards being scanned during a scavenge. During card table scanning, we only iterate over the dirty cards that correspond to the list of PABs for the task that initiates GC.

```
adjust_promotion_area(PromotionArea pa) {
 pa.start = adjust(pa.start);
 pa.end = adjust(pa.end);
 if(pa.start == pa.end) pa = NULL;
}
Word* adjust(Word* q) {
 if(q < first_dead) //GC maintains address of
        return;     //first dead object found in
                    //phase 2 of mark-compact
 new_q = NULL;
 while(q < end) { //end here is the end of old gen before GC
        new_q = forwaring_word(q);
        if(is_gc_marked(q)) {
                return new_q; //forwarding word is new location
        } else {
                if(new_q != NULL) {
                //fast case in determining next live object
                //q happens to be the first dead object of a
                //clump of dead objects: next live object is new_q
                q = new_q;
                } else {
                //q happens to be in the middle of a clump of dead
                //objects. Iterate till we find the next live object.
                q = q + size(q);
                }
        }
 }
 //we reached the end without finding the new location for q
 if(q > new_top)      //new_top is the end of the last live
         return new_top; //object after GC
 return NULL;
}
```

Figure 4. PAB adjustment at full GC. *pa* is the PAB to be adjusted.

3.3 Task-Concurrent Scavenging

By combining independent young generations and promotion area buffers, we implement a mechanism that enables mutator activity and minor collections to be performed concurrently. We refer to this mechanism as *mutator-concurrent scavenging*.

Mutator-concurrent scavenging requires maintaining consistency while scanning of the old generation during promotion. In order to maintain a consistent view of the old generation, changes to the old generation during direct allocation must not affect old generation objects accessed during scavenging. This requires that both object allocation and initialization of the object be done atomically in order for the collector to only trace objects with valid class information. Guaranteeing the atomic behavior of these two operations cannot be done efficiently with non-blocking synchronization (in contrast to allocation alone which can be implemented with a single compare-and-swap operation, i.e. `cas`). Other synchronization mechanisms would impose a prohibitive overhead on allocation.

PABs provide a synchronization-free solution since we need only to scan task-private PABs during scavenging. Other tasks may directly allocate in their own private PABs without affecting minor collection.

Key to mutator-concurrent scavenging is a modified synchronization mechanism that only pauses threads that belong to the task that triggers collection (the *trigger* henceforth), during scavenging. This process first obtains a global *Threads_lock* so that no new threads can be started, or existing threads terminated while the runtime is negotiating a safepoint. We then count the number of threads belonging to the trigger that are running, and iterate until this number reaches zero.

In the MVM, threads periodically poll (access) a constant reserved address that does not belong to the application heap. This address lies on a protected page and accessing this page results in an exception. The exception handler is responsible for blocking threads for a safepoint operation. We make polling task-aware by making threads access a task-private polling page. When a non-global safepoint is initiated, we set only the polling page for threads belonging to the trigger to an address that corresponds to a protected page. An exception will be triggered for these threads when they poll for a safepoint.

```
begin_per_task_safepoint {
  Threads_lock->lock(); //no threads should terminate or start
  Safepoint_lock->lock(); //only 1 safepoint at a time
  ∀ t ∈ Threads
        if wants_safepoint(t) { //t belongs to initiator
                ++running;
                protect(t.polling_page);
        }
  while(running > 0) {
         ∀ t ∈ Threads
                if wants_safepoint(t) {
                        //wait until t is waiting on
                        //Scavenge_lock
                        if(!is_running(t))
                                --running;
  }
  //safepoint reached
  Safepoint_lock->unlock();
  Threads_lock->unlock();
}

end_per_task_safepoint {
  Threads_lock->lock(); //no threads should terminate or start
  Safepoint_lock->lock(); //only 1 safepoint at a time
  ∀ t ∈ Threads
        if wants_safepoint(t) { //t belongs to initiator
                unprotect(t.polling_page);
                t->restart();
        }
  Scavenge_lock->notify_all(); //wake up all threads waiting
                               // on the Scavenge_lock
  Safepoint_lock->unlock();
  Threads_lock->unlock();
}
```

Figure 5. Per-task safe pointing. *begin_per_task_safepoint* initiates a safepoint for a single task and *end_per_task_safepoint* ends it and resumes mutators for that task.

The exception handler causes threads to wait on a *Scavenge_lock*, which will only be released when scavenging is complete. Note that only threads belonging to the trigger will wait on the *Scavenge_lock*. When all such threads are paused, the number of threads running drops to zero, and GC commences. Threads belonging to other tasks may continue to allocate, however, they may not perform a GC while the current GC is in progress. Releasing a safepoint is the reverse of this process. The private polling page for blocked threads is set to an address belonging to an unprotected page, and the *Scavenge_lock* is released. This process is illustrated in Figure 5.

4. Evaluation

To evaluate our extensions to MVM memory management, we performed a number of empirical experiments. We gathered our results using a dedicated machine equipped with two UltraSPARC^{TM} processors clocked at 1.5GHz, and running the Solaris^{TM} 10 Operating System (OS). The MVM implementation that we extended in this work is based on the Java HotSpot client virtual machine version 1.5. We present results for a number of SpecJVM98 [25] and Dacapo [9] benchmarks.

To evaluate the performance of our system, we first present throughput and response time for short running tasks, when executing concurrently with a GC-intensive program. We then consider throughput as well as the overall performance of concurrent, ho-

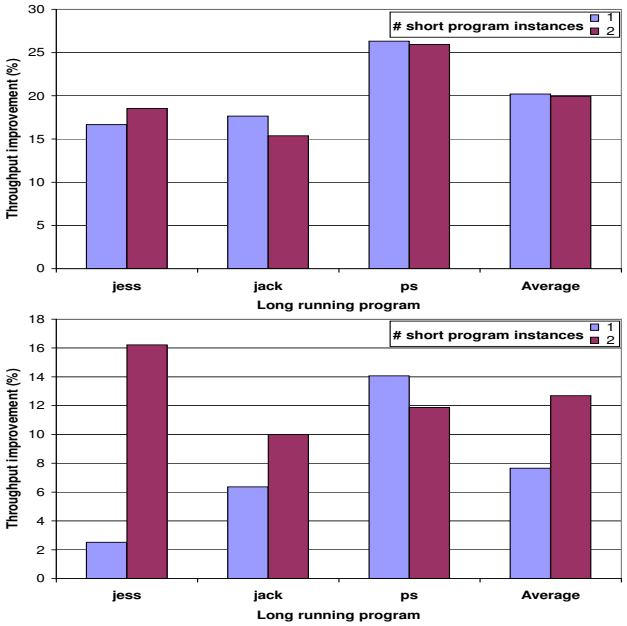

Figure 6. Throughput improvement enabled by independent young generations & PABs for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications: `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program.

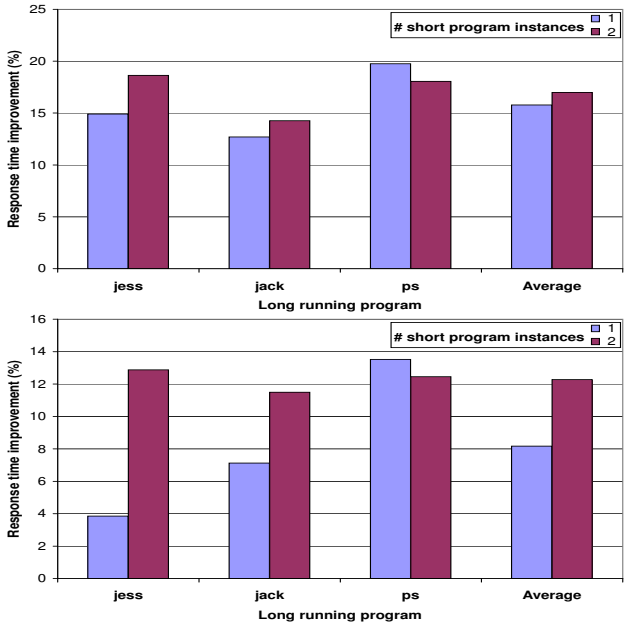

Figure 7. Response time improvement enabled by independent young generations & PABs for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications – `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program.

mogeneous tasks. Finally, we analyze the impact of our techniques on the number of GCs that the system performs as well as the time spent in GC.

In the first set of results (Figures 6 and 7), we show the throughput and response time improvement enabled by independent young generations and PABs over a system with a shared young generation. In this set of experiments we execute multiple serial instances of a short running application, concurrently with a single instance of a GC-intensive application in a fixed time interval. The goal is to measure the number of serial instances of the short running application that can be executed with the shared young generation system, versus the number of instances of the same application executed with an implementation that includes independent young generations and PABs. We also report response time (average application time) for the short running application. The goal of these experiments is to show the throughput increase (measured as the extra number of serial instances of the small application we can execute), and the response time improvement, of our system versus the shared young generation system. The short applications we consider are `javac` & `javap` with small command-line inputs (which we can provide on request), and the GC-intensive applications are `jess`, `jack` and `ps`.

The results show that in all cases, we enable a significant throughput increase and a response time improvement over a shared young generation system. For `javap`, on average, throughput improvement seems to increase with two concurrent short applications, over a single instance of that application. This is due to the fact that `javap` is very short running and does not exercise GC, and, two instances can be optimally scheduled on our two processor system. For `javac`, throughput gains remain almost the same with two concurrent instances since it does perform stop-the-world

Bmark	# GCs		Number of tasks									
			1		2		3		4		5	
	Minor	Major	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)
jess	146	2	4.59	302	6.17	608	9.58	1001	12.65	1346	16.60	1893
raytrace	76	2	2.82	257	3.76	533	5.74	765	7.25	900	8.98	1157
db	38	2	18.53	255	21.85	638	33.25	1000	43.89	1809	57.16	4164
mpeg	1	1	8.73	50	8.89	95	13.44	149	18.17	190	22.46	272
jack	99	8	4.16	649	5.39	939	7.64	1690	9.38	1706	14.17	2322
ps	217	0	26.67	118	43.96	477	57.67	817	74.59	1272	90.84	1878
jython	142	0	14.32	222	24.75	1408	32.73	2246	42.31	2785	51.22	3446

Figure 8. Data for the Base MVM system (shared new generation). Columns 2 & 3 show the number of minor (scavenges) and major collections respectively for a single instance of the benchmark in Column 1. The rest of the columns show execution time (ET) in seconds & GC time (GCT) in milliseconds for 1, 2, 3, 4 and 5 concurrent instances, respectively, of the programs listed. Figures 9 and 11 show improvement relative to this data.

GC. Figure 7 shows similar trends for the response time. Response time for `javac` is improved by over 15%, while, `javap` shows a 8% to 12% improvement.

In summary, the impact on the execution of a short running program that concurrently executes with another program that shows significantly heap usage, is visibly reduced. This is an effect of performance isolation provided by per-task young generations and fast tenured generation reclamation provided by PABs.

We next evaluate the overall performance of our mutator-concurrent scavenging system for a concurrent workload. Figure 8 shows data for the original MVM, which is configured with a shared new generation (we henceforth refer to this configuration as the *base*). This includes the number of minor and major GCs,

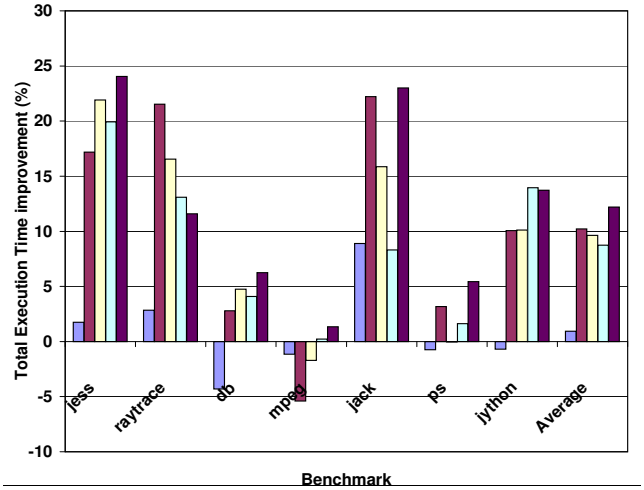

Figure 9. Total end-to-end performance improvement enabled by mutator-concurrent scavenging over the base MVM for homogeneous benchmark instances. Bars indicate increasing number of tasks (from 1 to 5).

Bmark	Change in # GCs									
	1		2		3		4		5	
	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major
jess	9	-2	18	-3	26	-4	34	-4	42	-4
raytrace	5	-1	9	-2	76	-1	95	-2	155	-1
db	2	-1	25	0	57	-1	105	-1	136	-4
mpeg	0	0	0	0	0	0	0	0	0	0
jack	6	-9	11	-9	16	-15	80	-11	26	-11
ps	14	0	25	-1	36	-1	48	-1	58	-2
jython	8	-1	16	-11	23	-15	31	-15	38	-16

Figure 10. Change in the number of GCs (minor and major) with mutator-concurrent scavenging over the base MVM for 1 thru 5 instances of the same benchmark.

total execution time and GC time for up to 5 concurrent homogeneous instances of the benchmarks.

Figure 9 shows the percent improvement in the end-to-end performance enabled by mutator-concurrent scavenging over the base MVM. The mutator-concurrent scavenging configuration includes the promotion area buffers (PAB) implementation. The bars represent homogeneous concurrent tasks, with one to five tasks (left to right bars).

Mutator-concurrent scavenging enables a 10-12% performance improvement for this configuration, across benchmarks on average. `jess` and `jack` show the most improvement (over 20% in many cases), since they involve significantly more GC activity compared to other benchmarks. `raytrace` also shows similar behavior. This is apparent from the data in Figure 8.

In the base system all tasks must pause for a minor collection triggered by any task, hence applications that cause more GC activity scale poorly. Although `ps` causes a large number of scavenges, the improvement is less pronounced as a percentage of the total execution time due to the fact that the program is long running (over a minute). We believe that `mpeg` does not make significant use of the heap and thus, does not reap the benefits from young generation isolation or concurrent allocation techniques. In fact, performance is slightly degraded for this benchmark due to an increase in GC time (explained below).

We next investigate the impact of our techniques on GC activity. Figure 10 shows the change in the number of scavenges and full GCs over the base MVM, for one to five concurrent homogeneous tasks, for each benchmark. We observe that with mutator-concurrent scavenging, the number of scavenges slightly increases

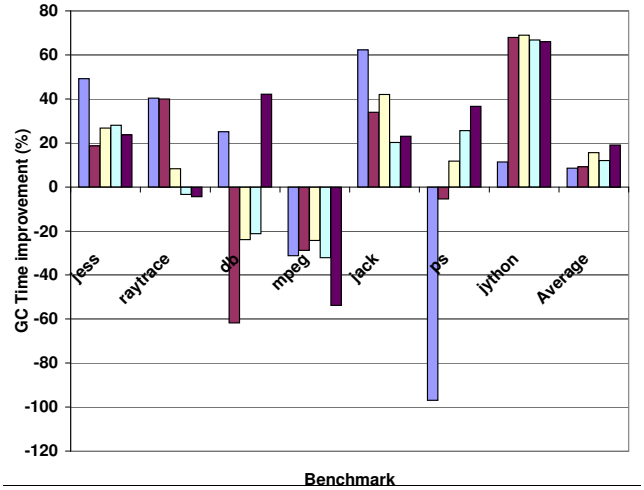

Figure 11. Total GC time improvement (minor + major) enabled by mutator-concurrent scavenging over the base MVM. Bars indicate increasing number of homogeneous tasks (from 1 to 5).

in a majority of the programs. The reason for this is that in the base system, a scavenge copies live objects from the entire young generation. Consequently, at the end of the scavenge, the young generation is empty. However, with mutator-concurrent scavenging, promotion is isolated and only the trigger's objects will be promoted. At the end of the scavenge, only one of the young generations will be empty, while the rest may yet trigger a GC since they are allocating independently. However, we perform less work during any single scavenge. Note that `mpeg` shows no change in the number of GCs.

Figure 11 shows the percentage change (improvement) in GC time for our implementation versus the base MVM. These figures show a reduction in full GCs with independent scavenging, and a consequent reduction in total GC time, ranging from 9% to 19%. Since, the mutator-concurrent scavenging configuration also includes PABs, as tasks terminate, other tasks start using the terminated tasks' freed PABs, thus leading to full GC avoidance. Full GCs are much more expensive than scavenges, hence, reduction in full GCs results in a sizeable reduction in overall GC time. Cases in which we are unable to avoid full GCs, do not show an improvement in GC time. In fact, time spent per garbage collection in our system is higher than the base MVM, leading to a performance degradation in case in which we don't reduce GCs. This is due to the extra time spent in iterating over a discontinuous set of PABs in old generation. This is especially visible in the case of `mpeg`, `db`, and `ps`.

Note, however, that GC time is not an indication of overall concurrent system performance.

To summarize, in the base system, every concurrent application will pause on every GC, and therefore experience degraded performance which independent scavenging significantly improves upon. Yet mutator-concurrent scavenging does not impact total GC time adversely although more scavenges are performed than with a shared new generation. Coupled with reclamation of promotion areas, mutator-concurrent scavenging reduces the number of full GCs, which are generally more expensive than scavenges, resulting in an improvement in total GC time in most cases.

5. Related Work

The techniques that we present herein build upon a body of related work in garbage collection. Other GC systems employ similar

collection strategies and old general reclamation mechanisms to those that we describe herein, however, in a different context.

A prior MVM implementation [7] includes per-task young generations and implements reclamation of young generations. It also describes temporary dynamic extension of the young generation space. It does not, however, provide reclamation of per-task old generation areas without triggering a full GC. This is especially important for non-trivial tasks that utilize the old generation. More importantly, we provide the ability to collect per-task young generations without pausing all tasks, which leads to better scalability. In addition, the prior work requires scanning of dirty cards belonging to all tasks during scavenging. This makes scavenge dependent on the number of tasks, which inhibits scalability. Our promotion allocation buffers provide precise tracking of regions of the old generation on a per-task basis, and the number of concurrent tasks. This allows us to only scan cards belonging to the GC trigger task.

Detlefs et al's garbage-first GC [12] splits the heap into regions which can be independently collected to satisfy a soft pause-time limit. The authors employ bidirectional remembered sets between regions to allow any set of regions to be collected independently of the others. They use GCLABs, which are thread private allocation buffers that they use during GC time. Since their collection policy is concurrent, threads compete to perform an object copy. Since we assign PABs on a per-task basis, in the common case, there is only one per-task thread performing promotion, without the need for synchronization.

A constraint imposed by our current implementation is the need to pause all tasks in order to perform an old generation collection. Our experience with MVM has been that stop-the-world full heap collections are relatively infrequent in a multi-tasking environment running short tasks, and of lesser importance for a system equipped with instantaneous and transparent heap reclamation upon task termination. Even though full GCs are less likely to be a significant performance bottleneck in our system, short running tasks may still be unnecessarily penalized if a longer running task triggers multiple full GCs. We are pursuing ways in which only regions of the old generation belonging to a particular task may be collected, independently and perhaps incrementally [18, 24], in order to further reduce the pauses experienced by tasks.

In addition to a stop-the-world major collection, we also only allow allocation and minor collection to occur concurrently. Our experience has been that the major source of performance bottleneck in a multi-tasking environment is the contention between mutators and GC, and less so between minor collections. However, introducing the ability to concurrently [4] collect young generations might enable further performance gains. We plan to consider such an implementation as part of future work.

Prior work on thread-specific heaps [14, 26] focuses on improving performance for an application by enabling garbage collection on a per-thread basis, to minimize synchronization between application threads. Although, this helps achieve performance isolation, our work is different in that there is no sharing of objects between tasks in MVM. Consequently, we can achieve better isolation since we do not need to track references between young generations. Thread-specific heap techniques can be combined with our scheme to provide further performance isolation. However, performance isolation constitutes only a part of our work. A significant goal is also to accurately identify heap usage, and readily reclaim heap space upon task termination, without requiring collection, minor or major.

Other researchers have presented complementary schemes for reducing old to young generation scanning time during minor collection, which may be combined with our per-task card table scanning mechanism. Azagury et al present a scheme for combining card marking with remembered sets [2, 16] in the train collector [18]. They maintain a per-card remembered set that is updated during card scanning, so that the card does not have to be scanned repeatedly unless it is modified. Another complementary approach for reducing scanning time, is to use a 2-level card table, with coarse and fine grain cards [11]. This is especially lucrative for large heaps, since regions of the heap that do not include old to young generation pointers can be logged as a few coarser level cards and quickly skipped.

Dimpsey et al [13] discuss compaction avoidance which leverages two key concepts [20] – address ordered allocation, and wilderness preservation. These techniques minimize heap fragmentation, and consequently, the frequency of compaction. Since our allocation scheme is a bump pointer, we automatically ensure address ordered allocation. Our scheme does not require free lists to be maintained and rebuilt by a mark phase. In addition, the part of the old generation beyond the end of the last PAB acts as a wilderness region. We perform large object allocations from this region directly (instead of PABs), thereby reducing fragmentation.

The KaffeOS [3] provides isolation and resource management for untrusted Java applications. The primary aim is to provide protection and isolate applications from each other, and to control resources on a per-application basis. The MVM concept of an isolate is a much lighter-weight abstraction than a KaffeOS process, hence more efficient, albeit with fewer features. The garbage collector in the KaffeOS is non-generational and conservative. Consequently, there is no provision to handle correctness and efficiency on a per-process basis in the presence of modern GC techniques, such as pre-tenuring, or thread-local allocation areas. Our work employs a state-of-the-art generational collector with each task using its own separate young generation, but with the old generation shared across tasks to enable better scalability. In addition, we enable optimizations, such as fast reclamation of old generation areas upon task termination, PABs and efficient card table scanning, in order to optimize throughput for modern multi-application environments.

Lastly, our old generation reclamation scheme does not require marking and tracing [21, 30] to identify per-task mature objects. By tracking promotion areas, we can readily reclaim all per-task dead mature objects upon task termination.

6. Conclusions

We present a series of generational memory management techniques to improve the efficiency and scalability of a multi-tasking virtual machine (MVM) for the Java programming language. Our techniques (i) partition the young generation into per-task regions that are isolated from other tasks, (ii) track old generation heap consumption on a per-task basis, and (iii) facilitate concurrent mutation activity with minor collection. These MVM extensions enable fine-grain control of task-specific heap parameterization and accounting, immediate reclamation of heap areas upon task termination, concurrent allocation in the young generation, promotion of objects during minor and major collection for only the task that triggers GC, and reduced scanning overhead during GC. Our results indicate that these benefits translate into significant improvements in efficient concurrent task execution and overall system throughput and scalability.

Acknowledgments

We thank Grzegorz Czajkowski at Sun Microsystems Laboratories for his extremely valuable input and suggestions on this work. We also thank the anonymous reviewers for providing useful comments for the final version of this paper. This work was funded in part by NSF grant Nos. CAREER-CNS-0546737, ST-HEC-0444412, and EHS-0209195.

Trademarks

References

[1] Multitasking virtual machine. `http://mvm.dev.java.net`.

[2] AZAGURY, A., KOLODNER, E. K., PETRANK, E., AND YEHUDAI, Z. Combining Card Marking with Remembered Sets: How to Save Scanning Time. In *International Symposium on Memory Management (ISMM)* (Oct. 1998).

[3] BACK, G., AND HSIEH, W. C. The KaffeOS Java Runtime System. *ACM Trans. on Programming Languages and Systems 27*, 4 (July 2005), 583–630.

[4] BARABASH, K., OSSIA, Y., AND PETRANK, E. Mostly Concurrent Garbage Collection Revisited. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2003).

[5] CHAMBERS, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, Mar. 1992.

[6] CZAJKOWSKI, G. Application Isolation in the Java™ Virtual Machine. In *OOPSLA* (2000), pp. 354–366.

[7] CZAJKOWSKI, G., AND DAYNÈS, L. Multitasking without Compromise: A Virtual Machine Evolution. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2001).

[8] CZAJKOWSKI, G., AND DAYNÈS, L. A Multi-User Virtual Machine. In *USENIX 2003 Annual Technical Conference* (June 2003).

[9] The Dacapo Benchmark Suite, version beta050224. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.

[10] DAYNÈS, L., AND CZAJKOWSKI, G. Sharing the Runtime Representation of Classes Across Class Loaders. In *European Conference on Object-Oriented Programming (ECOOP)* (July 2005).

[11] DETLEFS, D., CLINGER, W. D., AND JACOB, M. Concurrent Remembered Set Refinement in Generational Garbage Collection. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)* (Aug. 2002).

[12] DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. Garbage-First Garbage Collection. In *International Symposium on Memory Management (ISMM)* (Oct. 2004).

[13] DIMPSEY, R., ARORA, R., AND KUIPER, K. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal 39*, 1 (2000). `http://www.research.ibm.com/journal/sj/391/dimpsey.html`.

[14] DOMANI, T., GOLDSHTEIN, G., KOLODNER, E. K., LEWIS, E., PETRANK, E., AND SHEINWALD, D. Thread-Local Heaps for Java. In *International Symposium on Memory Management (ISMM)* (June 2002).

[15] HÖLZLE, U. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems* (Oct. 1993).

[16] HOSKING, A. L., AND HUDSON, R. L. Remembered Sets Can Also Play Cards. In *OOPSLA '93 Workshop on Garbage Collection and Memory Management* (Sept. 1993).

[17] HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIĆ, D. A Comparative Performance Evaluation of Write Barrier Implementations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 1992).

[18] HUDSON, R. L., MORRISON, R., MOSS, J. E. B., AND MUNRO, D. S. Garbage Collecting The World: One Car At A Time. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 1997).

[19] JAVA COMMUNITY PROCESS. JSR-121: Application Isolation API Specification. `http://jcp.org/jsr/detail/121.jsp`.

[20] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, Dec. 1997.

[21] MCCARTHY, J. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Comm. of the ACM 3* (1960), 184–195.

[22] MORRIS, F. L. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM 21*, 8 (1978).

[23] PALECZNY, M., VICK, C., AND CLICK, C. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)* (Apr. 2001).

[24] SACHINDRAN, N., ELIOT, J., AND MOSS, B. Mark-copy: Fast Copying GC with less Space Overhead. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2003).

[25] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[26] STEENSGAARD, B. Thread-Specific Heaps for Multi-Threaded Programs. In *International Symposium on Memory Management (ISMM)* (Oct. 2000).

[27] SUN MICROSYSTEMS INC. Java 2 Platform Standard Edition 5.0. `http://java.sun.com/j2se/1.5.0/index.jsp`.

[28] UNGAR, D. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Notices 19*, 5 (1984), 157–167.

[29] UNGAR, D. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Software Engineering Symposium on Practical Software Development Environments* (Apr 1992).

[30] WILSON, P. R. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management* (Sept. 1992).