To appear in Proc. 14th Workshop on Languages and Compilers for Parallel Computing

A Comparative Evaluation of Parallel Garbage Collector Implementations

Clement R. Attanasio David F. Bacon Anthony Cocchi Stephen Smith

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Abstract. While uniprocessor garbage collection is relatively well understood, experience with collectors for large multiprocessor servers is limited and it is unknown which techniques best scale with large memories and large numbers of processors. In order to explore these issues we designed a modular garbage collection framework in the IBM Jalapeño Java virtual machine and implemented five different parallel garbage collectors: non-generational and generational versions of mark-and-sweep and semi-space copying collectors, as well as a hybrid of the two. We describe the optimizations necessary to achieve good performance across all of the collectors, including load balancing, fast synchronization, and inter-processor sharing of free lists. We then quantitatively compare the different collectors to find their asymptotic performance both with respect to how fast they can run applications as well as how little memory they can run them in. All of our collectors scale linearly up to sixteen processors. The least memory is usually required by the hybrid mark-sweep collector that uses a copying collector for its nursery, although sometimes the non-generational mark-sweep collector requires less memory. The fastest execution is more application-dependent. Our only application with a large working set performed best using the mark-sweep collector; with one exception, the rest of the applications ran fastest with one of the generational collectors.

1 Introduction

This work reports the design, implementation, and performance of a family of garbage collectors that are part of the Jalapeño project, a Java Virtual Machine written in Java, targeted mainly for large symmetric multiprocessor (SMP) machines executing server applications.

These garbage collectors are type accurate both in the heap and thread stacks, stop-the-world (synchronous with respect to mutator execution), and load-balancing among all available processors. We implemented both mark-and-sweep and semi-space copying collectors, with non-generational and generational variants of each. We also implemented a hybrid collector that uses mark-and-sweep for the mature space and copying for the nursery.

In this paper we describe the novel features of our implementations and quantitatively compare their performance. We show how varying heap size affects application speed and how the collectors scale as the number of processors is varied up to 16.

We observe a near-linear increase in throughput and decrease in delay time for garbage collection as the number of processors increases. Scalability is primarily limited by the size of the live heap; for large-scale applications like `SPECjbb` we observe linear scalability up to 16 processors; for others with only one or two mutators and small working sets scalability is limited to 5-8 processors.

These results were obtained by careful avoidance of delays due to load imbalance or excessive synchronization among the processors. We build work lists of objects to be scanned in buffers private to each processor, and then scan them in parallel, with the ability to exchange buffers among processors for load balancing. These techniques are similar to those described by Endo et al [7].

Jalapeño creates and manages Java threads explicitly. These threads are suspended for multiprogramming only at garbage collection safe points, so when garbage collection is triggered, all processors can begin collection as soon as the mutator thread in execution on each processor reaches a safe point. All non-executing threads are already at safe points.

In Jalapeño, everything is a Java object, including the objects that implement the JVM itself, such as thread stacks, the compiled code, and the collectors, and this requires some special considerations.

The rest of this paper is organized as follows: in Section 2 we describe the overall structure of Jalapeño and how garbage collection is integrated into the system. In Section 3 we describe the garbage collectors. In Section 4 we describe our benchmark programs and results. In Section 5 we discuss other work related to parallel garbage collection. Finally, we summarize our conclusions.

2 The Jalapeño Architecture and Garbage Collector Framework

In this section we describe the important features of the system in which we implemented the collectors studied in this paper, and describe the design choices that allow modular combination of different collectors and compilers.

Jalapeño [2] is a Java Virtual Machine for large symmetric multiprocessor (SMP) servers currently under development at the IBM T.J. Watson Research Center. Jalapeño is written in Java extended with unsafe primitives to allow direct manipulation of memory [1]. This simplifies some aspects of allocator and collector implementation, and complicates others.

Jalapeño uses the following well known-techniques to achieve high performance:

$M \times N$ **threading.** Java threads are multiplexed onto operating system threads, and Jalapeño schedules and migrates Java threads across operating system threads as needed. We call the operating system threads *virtual processors* (VPs).

Processor-Local Allocation. To avoid synchronization on every allocation, VPs allocate small objects out of larger chunks obtained from a central storage manager.

Safe Points. To ensure that threads are in a known state when garbage collection occurs, threads are only interrupted at safe points, for which the compiler generates maps that describe the contents of the stack. Jalapeño generates safe points at method calls and backward branches.

Type Accuracy. The combination of class descriptors and stack maps for safe points allows Jalapeño to perform type-accurate GC, meaning that it can locate all pointers in the system and find their correct run-time types.

Inlined Allocation. While many systems inline allocation code, in Jalapeño it occurs naturally because the allocation routines are also written in Java. Therefore, the general-purpose optimizer can inline allocation (system code) into the allocating methods (user code) and perform constant folding and other optimizations. When the type of an allocation is known, its size and finalization status are known, and two conditional branches can be eliminated from the allocation code.

While these are all well-known techniques, the synergy between the design choices is important. For instance, most Java Virtual Machines allocate one operating system thread per Java thread. As a result, it can take a long time to perform a barrier synchronization, since each thread must either wake up and come to some sort of safe point, or be put to sleep by the collector. Both approaches are costly, and the cost is proportional to the number of threads instead of the number of processors; therefore, these approaches do not scale. In addition, if the latter approach is taken, then the threads will be suspended at unknown points, making type-accurate collection difficult or impossible.

2.1 Modular System Design

Jalapeño supports multiple compilers and multiple collectors (as have other modern virtual machine designs [8, 14]). The different compilers can co-exist within a single virtual machine instance; the collector must currently be selected when the virtual machine image is generated.

There are three different compilers: *Baseline*, *Quick*, and *Opt*. The Baseline compiler performs direct translation of bytecodes into native machine instructions, simply emulating the virtual machine stack (there is no interpreter, so the Baseline compiler serves in place of an interpreter). The Quick compiler performs some simple optimizations, primarily register allocation. The Opt compiler is a full-blown multi-pass optimizing compiler. In addition, optimization levels can be dynamically controlled at run-time through feedback-directed optimization [3].

Each compiler generates `MapIterator` objects that are used to iterate over the stack frames it generates and find all of the pointers in each frame. A stack can contain a mixture of frame types generated by different compilers. The map iterator presents a common compiler independent interface used by all the collectors when scanning stacks for roots.

3 The Garbage Collectors

3.1 Storage Layout

Figure 1 illustrates how the Jalapeño address space is managed.

Boot Image. The boot image contains the memory image, generated by the build process, with which Jalapeño begins execution. As a minimum it contains the Java

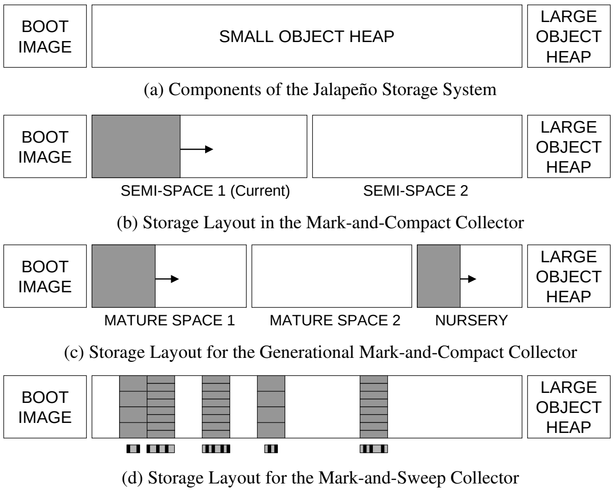

Fig. 1. Garbage Collector Storage Layout

objects necessary for bootstrapping the complete Jalapeño virtual machine. Optionally, e.g., for faster startup, it can contain more than the minimal set. For example, the Jalapeño optimizing compiler can be produced at build time and included in the boot image.

Small Object Heap. All collectors divide the small object heap into "chunks," typically 32KB, and allocate chunks as required to virtual processors, so that small objects can be allocated without synchronization within chunks. When needed, new chunks are obtained in a synchronized fashion from a global pool. When no more chunks are available, a garbage collection is triggered.

Semi-space Copying Collector. Given a chunk, the semi-space storage manager maintains a next pointer from which to satisfy a request. If the requested size does not extend beyond the chunk boundary, the next address is returned, the next pointer is updated to the end of the newly allocated space, and execution continues.

Mark-sweep Collector. The mark-sweep storage manager divides a chunk into fixed size slots. Currently it defines (at system build time) twelve slot sizes ranging from 12 bytes to 2KB. One chunk for each slot size is allocated to each virtual processor. A request is satisfied from the current chunk for the smallest slot size that can contain the requested size. Available slots in each chunk are maintained in a singly-linked list, and each virtual processor has a set of pointers to the next available slot for each slot size. Allocation involves updating the next pointer for the requested size for the given virtual processor. When a next pointer is zero, then a new chunk (either previously allocated or obtained from the chunk manager) becomes the current one for this virtual processor, and the list of free slots in the chunk is built.

Allocation. In the usual case when the allocation is satisfied in the current chunk, an optimized, inlined allocation executes ten instructions for the copying collector, and seventeen for the mark-sweep collector.

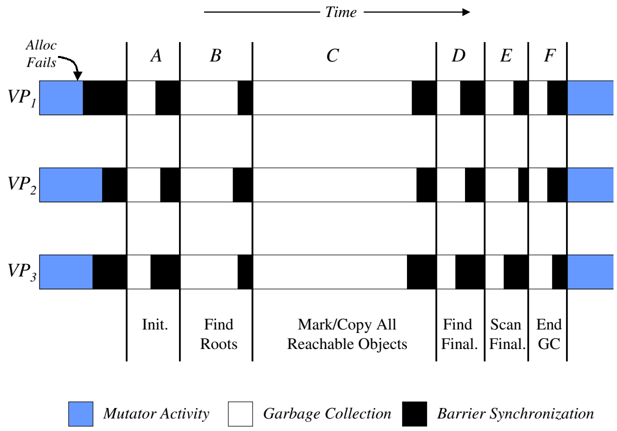

Fig. 2. Phases of Parallel Garbage Collection.

Large Object Heap. All collectors share the same large object manager. Large objects (larger than 2KB) are allocated using a first-fit strategy in contiguous 4KB pages. They are never moved during garbage collection. As large objects become garbage, their pages are automatically coalesced with adjacent free pages.

3.2 Parallel Collection

When an allocation request cannot be satisfied, all virtual processors are notified that a garbage collection is required. The mutator thread executing on each virtual processor, at its next garbage collection safe point, transfers execution to the Jalapeño dispatcher which suspends it and schedules the garbage collection thread. Recall that all other suspended mutator threads are already at safe points.

Coordination among executing garbage collection threads is implemented by barrier synchronization. Figure 2 illustrates the phases of garbage collection, separated by barrier synchronizations.

In phase A, initialization and collector-specific housekeeping functions are performed. The mark-sweep collector uses a side array associated with each allocated chunk for marking live objects; these must be zeroed during initialization.

The Jalapeño Table of Contents (JTOC) contains pointers to all class objects and static fields of both the Jalapeño virtual machine and applications. These, and pointers in the execution stacks of programs, comprise the roots for garbage collection. Pointers in stacks are found by the exact stack mapping mechanism implemented by all Jalapeño compilers. To accommodate relocation of code objects and execution stacks, copying collectors require the barrier after Phase B, to ensure that such objects are processed with special logic first. Mark-sweep collectors do not require barrier B because they do not move objects.

Work Buffers. All Jalapeño garbage collectors are carefully engineered to execute in parallel by using local work buffers that contain object pointers to process. Root pointers are collected into such buffers. As a processor fills a buffer, it is added to a global list of work buffers. In Phase C, each processor repeatedly obtains a work buffer from the global list of work buffers, and for each object pointer in it not yet marked, marks, and if copying, copies it to “to” space, and scans it (with exact information) for pointers to other objects. Each pointer thus found is entered locally into its work buffer, which, when full, is added to the global list of work buffers.

Copying collectors must test and mark objects atomically because it would be an error for an object to be copied by more than one collector thread during a collection, leading to two copies of an object. Mark-sweep collectors do not mark atomically, since the rare race condition leads only to slight inefficiency of multiple scan of an object. We feel avoiding a hardware synchronizing instruction more than gains back the time for the occasional duplicate scan of an object.

In copying collectors, each collector thread obtains a chunk of “to” space from the global chunk manager, and then copies live objects into it without synchronization.

When there are no more work buffers in the global list, Phase C is complete.

Finalization. All Jalapeño collectors support finalization. When an object is allocated, if its class has a finalize() method (this is checked at compile-time rather than run-time) an element identifying it is entered on a “live and finalizable object” queue. This queue, if not empty, is scanned in Phase D, and if any objects found there have become garbage, they are made live, moved to a “to be finalized” queue, and treated as roots for a mark or mark/copy phase E. Finalize methods are executed by a distinguished mutator thread after collection completes, i.e., not as part of garbage collection.

If the “to be finalized” queue is empty after phase C, phases D and E are skipped, avoiding two barrier synchronizations. Thus the mark-sweep collector usually requires four barrier synchronizations, and the copying collector usually requires five.

Storage Reclamation. For all Jalapeño collectors, reclaiming storage involves little additional explicit work. For semi-space copying collectors, the previous allocation space is now available for reuse. For mark-sweep collectors, unmarked entries in the side array for each chunk identify available space. The side arrays must be scanned to find chunks containing no live objects, which are returned to the global chunk manager.

3.3 Generational Collection

Jalapeño also contains generational variants of both the semi-space copying and mark-sweep collectors. For the copying collector, the age of the object can be determined from its address. For the mark-sweep collector, it is kept in the associated side array.

The advantage of generational collection is that old objects are not scanned unless they may contain pointers to young objects. Additionally, in copying collectors, old objects are not copied. Thus the potential advantage of generational over non-generational collection is greater for copying collectors, since these avoid the cost of copying old objects repeatedly.

Jalapeño compilers provide a write barrier for old objects: the address of the old object stored into is written into a write buffer on the first store of an address during a mutator execution cycle. During the subsequent garbage collection, the write buffer is treated as another source of root pointers.

Program	Description	Code Size	T	Allocation			Smallest		Fastest	
				Objs	Total	HWM	Heap	Coll.	Time (s)	Coll.
213.javac	Java bytecode compiler	561 KB	1	8.2 M	294	9.9	22	MS	14.95	CG.2
227.mtrt	Multithreaded raytracer	571 KB	2	6.7 M	165	7.9	14	H.2	5.78	CG.8
228.jack	Parser generator	131 KB	1	8.1 M	331	2.4	13	H.2	19.09	CG.8
SPECjbb	TPC-C for Java	138 KB	8	7.9 M	280	148	180	H.16	[21933]	MS
jalapeño	Jalapeño optimizer	1378 KB	1	52.6 M	1433	11	34	H.4	25.8	CP
gcbench	GC Benchmark	4 KB	1	15.4 M	378	6.3	20	MS	4260	H.8

Table 1. Benchmarks and their overall characteristics. T is the number of simultaneous threads. Other quantities are MB of storage unless specified. Since `SPECjbb` is a throughput-oriented benchmark, we give its throughput score instead of the run time. HWM is the high-water-mark of allocation.

Hybrid Collector. Jalapeño also contains a hybrid collector, in which new objects are allocated in a nursery using the copying manager's strategy, but when objects survive a garbage collection, they are copied into an old space managed by the mark-sweep collector. This collector combines the faster allocation of the copying strategy with the avoidance of copying an object more than once. Most object allocations are performed with the copying algorithm. Most live objects are assumed to be old objects, and may be repeatedly marked but are not copied after they become old.

4 Performance Measurements

For the performance measurements we used six programs; four are part of the SPEC benchmark suite, `213.javac`, `227.mtrt`, `228.jack`, and `SPECjbb`. The remaining benchmarks are (`jalapeño`), the Jalapeño optimizing compiler compiling a set of some 740 class files into machine executable code, and the Java version of the `gcbench` benchmark, obtained from Hans Boehm's web site. While only `mtrt` and `SPECjbb` are parallel applications, the degree of parallelism of garbage collections was determined by the number of processors. For runs where the number of processors was not varied, eight processors were used.

All tests were run on a 24 processor IBM RS/6000 Model S80 with 50 GB of RAM. Each processor is a 64-bit PowerPC RS64 III CPU running at 450 MHz with 128 KB split first-level caches and an 8 MB unified L2 cache.

All benchmarks were run using the Jalapeño optimizing (*Opt*) compiler with the default level of optimization. All runs used the same size large object heap for each application. Since all collectors use the same large object heap code, we will not discuss large heaps in this section.

4.1 Benchmarks

The benchmark programs we measured are summarized in Table 1. For each benchmark, we show the code size (KB of class files), number of simultaneous threads (T),

Program	Minimum Heap Size						Std. Heap		Performance at Std. Heap Size					
	MS	CP	CG.S	CG.L	H.S	H.L	Total	Minor	MS	CP	CG.S	CG.L	H.S	H.L
213.javac	22	26	23	30	22	23	50	2/8	16.0	20.6	14.9	15.5	15.6	15.2
227.mtrt	17	18	19	24	14	18	50	2/8	6.50	6.21	5.84	5.78	6.03	5.91
228.jack	16	13	14	20	13	17	50	2/8	20.8	19.5	19.4	19.1	19.2	19.1
SPECjbb	240	280	300	400	180	240	400	16/64	21933	16611	18927	17216	18500	19892
jalapeño	38	34	46	60	34	44	80	4/16	30728	25811	28342	27257	28374	27212
gcbench	20	28	20	28	24	28	36	2/8	5642	4601	4907	4385	4835	4260

Table 2. Minimum heap sizes and performance for standard heap size when run with 8 CPUs.

allocation behavior, standard heap sizes, and summarize the best performance in both time and space.

Allocation behavior is broken down into three quantities: number of objects allocated (in millions), number of megabytes allocated, and high-water-mark (HWM) of bytes allocated. The latter number shows the largest live heap size that we observed.

Throughout the measurements, different collectors are labeled **MS** for the mark-and-sweep collector, **CP** for the copying semi-space collector, **CG** for the generational copying semi-space collector, and **H** for the hybrid collector that uses copying for the minor heap and mark-and-sweep for the major heap. For the CG and H collectors, they are suffixed by the size of the heap used. Generically, CG.S refers to the copying generational collector using the small heap size. For a particular run, the size is included, for instance `SPECjbb` run under the copying generational semi-space collector with a 16 MB nursery is labeled "CG.16".

Another important factor is that while we have tried to use the same benchmarks as other researchers (in particular `javac`, `SPECjbb`, and `gcbench`), some measurements will not be comparable due to the effects of our virtual machine being written in Java. In particular, the number of objects allocated includes objects that are allocated as part of the process of running the benchmark.

4.2 Absolute Performance

Table 1 provides an overview of our results on the absolute performance of the benchmarks with different collectors. We show both the minimum size in which the application ran, and the associated collector ("Smallest") as well as the best run in terms of completion time or throughput ("Fastest") when run with the standard heap size.

The results show that the hybrid collector with a small nursery (H.S) is most likely to run the applications in the smallest heap space. There are two reasons for this: first of all, the use of the nursery causes many short-lived applications to be collected quickly, so they never impact the space consumption of the mature space. Second, since the mature space uses the mark-sweep collector, the mature space is no divided into two semi-spaces. Put another way, the copying collector causes extremely coarse-grain fragmentation (always wasting a factor of two), while in the mark-sweep collector fragmentation is more localized and (in practice) limited.

Program	Number of Major Collections						Average Major Collection Time						Maximum Major Collection Time					
	MS	CP	CG.S	CG.L	H.S	H.L	MS	CP	CG.S	CG.L	H.S	H.L	MS	CP	CG.S	CG.L	H.S	H.L
213.javac	26	47	14	13	4	4	81	83	82	79	109	101	109	106	119	118	112	112
227.mtrt	11	18	3	3	0	0	71	74	67	72	—	—	86	93	71	85	—	—
228.jack	24	34	4	4	1	1	51	50	68	67	58	59	62	58	74	69	58	59
SPECjbb	36	98	53	46	11	11	243	359	389	388	260	265	305	396	424	427	278	281
jalapeño	64	94	29	26	16	13	68	72	106	108	89	99	109	118	212	160	142	150
gcbench	15	21	5	3	3	2	45	47	66	82	56	56	57	113	71	118	58	57

Table 3. Average and worst times for major collections in milliseconds when run with 8 CPUs. See Table 2 for standard heap sizes used.

For two benchmarks, `javac` and `gcbench`, the mark-sweep collector used less space. This is not surprising for `gcbench`, since it is a synthetic benchmark: there is a small number of object sizes, so fragmentation is almost zero, and the mark-sweep collector does not lose any memory to the nursery. We are still investigating `javac`.

Table 1 shows that for application speed, performance varies much more. In the case of `SPECjbb`, the best performance is obtained with the mark-and-sweep (MS) collector, which is due to the fact that it has a large working set of about 135 MB of live data, which the semi-space collectors must copy on each major collection. The hybrid collector achieves almost the same speed with a large nursery (H.64) — the gains from less object scanning seem to be offset almost exactly by the cost of the write barrier.

All but one of the other benchmarks achieved their best performance with some form of generational collector. This is due to a high allocation rate (which benefits from the cheaper allocation sequence in copying collectors) with a small working-set size (so that the expensive step of copying is minimized).

The SPEC benchmarks `javac`, `mtrt`, and `jack` all do best with the copying generational (CG) collector. For these benchmarks, the cost of the write barrier and the minor collections is gained back because fewer major collections are performed. On the other hand, `gcbench` does best with the hybrid (H) collector, probably due to its synthetic nature (as described above).

The `jalapeño` benchmark does best using the copying semi-space collector (CP), which has the lowest overhead when memory is plentiful.

In general, our results can be interpreted as strengthening the case of Fitzgerald and Tarditi [8] for profile-based selection for garbage collector selection.

Table 2. Graceful degradation.

The standard heap size is what we use for subsequent performance measurements, except those where the heap size was varied. These heap sizes were arrived at by finding a size in which each benchmark could comfortably run in all of the collectors. “Total” is the total heap size, including space for the minor heaps, if any. “Minor” gives two minor heap sizes, a small size (S) and a large size (L) for each benchmark.

Program	No. of Minor Collections				Avg. Minor Coll. Time				Max. Minor Coll. Time			
	CG.S	CG.L	H.S	H.L	CG.S	CG.L	H.S	H.L	CG.S	CG.L	H.S	H.L
213.javac	396	100	394	100	11	25	11	25	41	55	37	55
227.mtrt	184	45	184	45	6	11	6	11	33	41	32	41
228.jack	385	96	384	96	5	9	5	9	56	72	56	66
SPECjbb	464	108	454	122	32	67	32	68	58	90	56	91
jalapeño	948	236	943	235	20	37	19	35	220	278	196	200
gcbench	181	45	181	45	4	6	4	6	15	52	16	56

Table 4. Average and worst times for minor collections in milliseconds when run with 8 CPUs. See Table 2 for standard heap sizes used.

4.3 Major and Minor Collection Times

Table 3 shows the number of major collections, and average and worst-case pause times for the different collectors running with eight processors.

Interestingly, there is not a large amount of variation in average collection time. The greatest variability is in the number of major collections. The mark-and-sweep collector (MS) performs at least a third fewer collections than the copying collector (CP), which indicates that space loss due to semi-spaces is significant.

As expected, generational collectors greatly reduce the number of major collections; in one case to zero (`mtrt` with the hybrid (H) collector).

Table 4 shows the number of minor collections, and the average and worst-case pause times for the different collectors running on eight processors. Since the minor heap strategies are the same for both the copying semi-space generational (CG) and hybrid (H) collectors, the total number of minor collections is always roughly the same.

The minor collection times for hybrid collector are consistently slower because the cost of allocating in the mark-and-sweep collector that the hybrid (H) uses for the mature space is longer, while the copying generational (CG) collector uses a simple bump-and-test algorithm.

Note that even `SPECjbb`, with a live data size of about 135 MB and a 400 MB heap only experiences a 400 ms worst-case pause time for major collection and a 61 ms worst-case pause time for minor collection.

In general, the average and worst-case pause times for minor collections are very good, competitive with many times reported by "on-the-fly" collectors. Since the number of major collections is quite small, this may be acceptable for many applications.

4.4 Detailed Analysis of `SPECjbb`

The tables we have shown so far give a good overview of a number of different applications, and how they perform under various garbage collectors. In this section we will examine the performance of the largest and most parallel benchmark, `SPECjbb`, in considerably more detail.

Some previous work has only investigated collector speedups, and while this may be academically interesting, it provides no inherent benefit to the user. So the first question to answer is: what effect does varying collectors have on application performance?

Figure 3 shows how the throughput of `SPECjbb` with a 400MB heap as the number of processors is varied from 1 to 16.

The workload consists of 8 warehouses, so after 8 processors, there is no additional parallelism in the application. The graph thus provides an interesting view of both application and collector scaling.

Two collectors, mark-sweep (MS) and the hybrid with a 64 MB nursery (H.64), perform best across the entire range of configurations. They also scale extremely linearly, while the other collectors all show a certain amount of attenuation. In fact, `SPECjbb` has better throughput with 8 processors under MS or H.64 than it does with any other collector with 16 processors.

A different view of the collectors is shown in Figure 4, in which the number of processors is fixed at 8 and the size of the heap is varied from 1GB down to the minimum in which the application can run. Once again, mark-sweep and hybrid appear to be the best collectors, but there is a distinct crossover point between them. When memory is plentiful, the mark-sweep collector considerably outperforms all others (the reason will become clear when as we example subsequent data). Below 350MB, the mark-sweep collector begins to thrash, while the hybrid collectors defer a lot of major collections because short-lived objects are not promoted into the mature space. In general, we observe that generational collectors degrade much more gracefully in the presence of limited memory resources.

A crucial point here is that `SPECjbb` achieves the same performance under the hybrid collector with an 8MB nursery in a 220MB heap as the mark-sweep collector does with a 350MB heap. Therefore, if memory resources are limited (as they often are for transaction processing systems under heavy load), the hybrid collector may be able to deliver 50% better application throughput than the next best alternative.

Some clues as to what is happening are in Figure 5, which shows the total amount of time spent in garbage collection for `SPECjbb` with a 400MB heap for 1 to 16 processors. Two things are clearly visible in this graph. First, copying semi-space collectors spend much more time in garbage collection because they must, in general, perform twice as many collections as the comparable mark-sweep collector.

Generational collection significantly speeds up copying collection because the number of mature space collections drops significantly. However, in the case of mark-sweep collection, the generational hybrids (H.16 and H.64) are slower because they must perform more frequent minor collections and they also have the overhead of a write barrier, which the simple mark-sweep collector (MS) does not.

Figure 6 shows the average time for a mature-space collection as the number of processors is varied. This shows both the relative costs of collection under the different collectors, and their scaling properties. The copying collectors (CP and CG) take significantly longer per collection (as well as performing more collections due to use of semi-spaces) than the mark-sweep collectors (MS and H). In addition, the generational variants are slower because the nursery must first be collected.

A similar graph (not shown) for nursery collection shows that collection time is dependent almost entirely on the size of the nursery; this is not surprising since the same code is used for the nursery in both the copying generational (CG) and hybrid

(H) collectors. The hybrid collectors are slightly slower performing nursery collection, probably due to the more expensive allocation path in the mark-sweep mature space.

For `SPECjbb`, which generates large amounts of data on parallel threads, garbage collection scales well, even for nursery collection with small heaps. However, for other applications in our benchmark suite which are single-threaded and have a small working set, parallel collection of the nursery only scales to about 5 processors.

5 Related Work

Research in applying parallelism to garbage collection has fundamentally focused on two issues: response time and throughput. To optimize throughput, the collector is run in *parallel* across multiple processors, usually (but not always) while the mutators are stopped. To optimize response time, the collector is run *concurrently* with the mutators, and may or may not itself be multi-threaded.

Throughput-oriented parallel collectors have generally received less attention from the research community, probably because they do not offer any solution to the fundamental problems that garbage collection poses to time-sensitive computations. The fundamental tracing operation parallelizes fairly straightforwardly; important research questions have to do with load balancing and minimizing synchronization overhead.

Early work on parallel collection includes Halstead's implementation of a parallel copying collector for Multilisp [10] running on a 32-CPU Concert multiprocessor. His algorithm had each processor copying objects from any from-space into its local to-space.

Halstead's collector suffered from serious load-balancing problems. A similar approach was used by Crammond in the JAM Parlog collector [6]. Küchlin [13] presents a variant of Crammond's algorithm that trades time for space by using a divide-and-conquer technique that may require as many phases as there are processors but eliminates the need for tables containing inter-segment references.

Herlihy and Moss [11] describe a lock-free variant of Halstead's algorithm, but it requires multi-versioning which may significantly increase both space and time consumption.

Endo, Taura, and Yonezawa [7] implemented a parallel collector for a variant of C++ on a 64-processor Sun server. Their collector is based on the conservative Boehm collector [5]. They found that floating point numbers were often misidentified as pointers in one benchmark. Their work demonstrated that load-balancing could greatly improve scalability, and that large objects must be handled specially to maintain load balance. They also described how careful engineering of synchronization requirements improves overall performance. Unfortunately, although run on a real machine they only provide abstract speed and speedup measurements, without end-to-end performance numbers.

Recently, Boehm [4] has applied some of the techniques of Endo et al to create a version of his conservative collector that scales to small-scale (up to four CPU) multiprocessors that are becoming increasingly common, while still performing well on a single processor. He observes significant variation across benchmarks in the scalability of the collector, but also notes that due to excessive synchronization in `malloc()`, the garbage collected version often runs faster than the version that uses explicit allocation and de-allocation.

Most recently, Flood et al [9] describe parallelizing both a copying and a mark-compact collector. They compare the relative performance of the parallel versions with their serial counterparts. We have used two of the same benchmarks (`javac` and SPECjbb with 8 warehouses) for comparison. Our load balancing strategy is simpler, but seems to scale at least as well as theirs. Our scalability for `javac` is better, probably because we do not need to fragment the mature space for the compaction algorithm as they do.

6 Conclusions

We have shown through implementation and measurement that a variety of parallel collectors can be made scalable by careful attention to detail since all achieved near linear reduction in both major and minor collection times. The collectors can scale linearly up to 5-8 processors even when there is only a single mutator thread with a relatively small (10 MB) working set.

Performance of the collectors is very good: with eight processors we are able to collect 150 MB of live data from a 1 GB heap in 350 milliseconds.

The mark-and-sweep collector performs better when memory is tight or applications have a large working set as with the `SPECjbb` benchmark. Multiprocessor performance of the mark-and-sweep collector is crucially dependent on inter-processor sharing of free lists, which greatly reduces fragmentation, cuts collections by a factor of two, and allows applications to run in a considerably smaller heap.

Copying collectors clearly perform better when the creation rate of objects is high, and their lifetimes are short.

Generational collectors may be indicated when the creation rate of objects is high and at least some of them are long-lived, i.e., the working set is large.

When resources are abundant, there is no clear winner in application speed. However, when memory is limited, the hybrid collector (using mark-sweep for the mature space and semi-space copying for the nursery) can deliver at least 50% better application throughput. Therefore this collector seems best for online transaction processing applications.

References

[1] ALPERN, B., ET AL. Implementing Jalapeño in Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1999). *SIGPLAN Notices*, *34*, 10, 314–324.

[2] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (2000), 211–238.

[3] ARNOLD, M., FINK, S., GROVE, D., M.HIND, AND SWEENEY, P. Adaptive optimization in the Jalapeño JVM. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 2000). *SIGPLAN Notices*, *35*, 10.

[4] BOEHM, H.-J. Fast multiprocessor memory allocation and garbage collection. Tech. Rep. HPL-2000-165, Hewlett-Packard Laboratories, Dec. 2000.

[5] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Software – Practice and Experience 18*, 9 (1988), 807–820.

[6] CRAMMOND, J. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming 17*, 6 (1988), 497–522.

[7] ENDO, T., TAURA, K., AND YONEZAWA, A. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proc. of Supercomputing '97* (San Jose, California, Nov. 1997), IEEE Computer Society Press, Los Alamitos, California.

[8] FITZGERALD, R., AND TARDITI, D. The case for profile-directed selection of garbage collectors. In ISMM [12], pp. 111–120.

[9] FLOOD, C. H., DETLEFS, D., SHAVIT, N., AND ZHANG, X. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Usenix Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr. 2001).

[10] HALSTEAD, R. H. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

[11] HERLIHY, M., AND MOSS, J. E. B. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 3*, 3 (May 1992).

[12] *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), *SIGPLAN Notices 36*, 1.

[13] KÜCHLIN, W. A space-efficient parallel garbage collection algorithm. In *Proceedings of Fifth ACM International Conference on Supercomputing* (June 1991), pp. 40–46.

[14] PRINTEZIS, T., AND DETLEFS, D. A generational mostly-concurrent garbage collector. In ISMM [12], pp. 143–154.

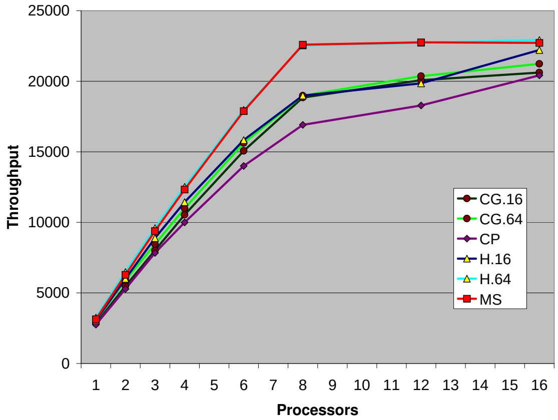

Fig. 3. Throughput for 8 clients as number of processors varies from 1 to 16.

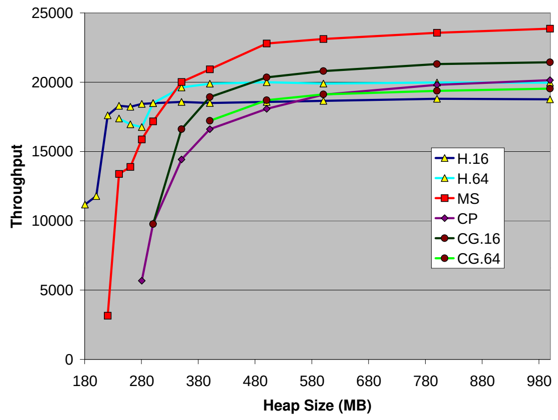

Fig. 4. Effect of heap size on application throughput with 8 processors (SPECjbb benchmark).

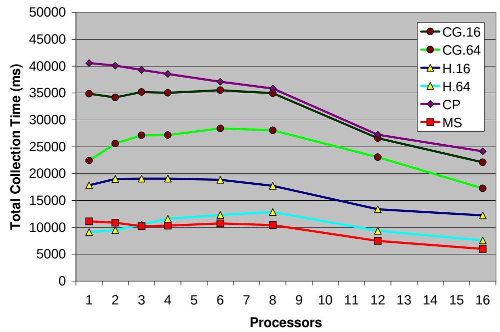

Fig. 5. Total garbage collection time (SPECjbb benchmark with a 400 MB Heap).

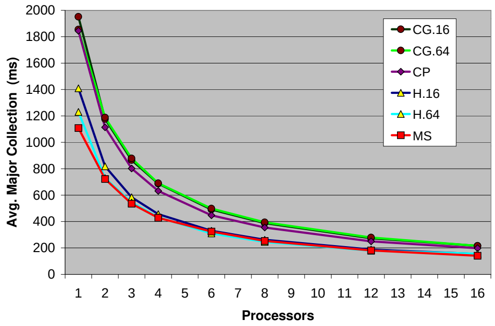

Fig. 6. Scalability of mature space collection (SPECjbb benchmark with a 400 MB Heap).