Behavior Characterization and Performance Study on Compacting Garbage Collectors with Apache Harmony

Chunrong Lai **Ivan T Volosyuk** **Xiao-Feng Li**

Enterprise Solutions Software Division, Software and Solutions Group, Intel Corporation

{chunrong.lai, ivan.t.volosyuk, xiao.feng.li}@intel.com

Abstract:

Compacting garbage collector (GC) has been recognized for its benefits in allocation efficiency and space utilization. Some commercial runtime systems are known to have compacting GC implemented [1][2]. Apache Harmony [3], an open source Java SE implementation, also has two versions of compacting GC built in, which were developed based on classical compaction algorithms. Our experiments with Harmony compacting collectors showed that the delivered performance is sensitive to the underlying platforms. A better performing collector in one platform can perform worse in another. In this paper, we evaluated Harmony compacting collectors in modern platforms with standard server benchmark, characterized the collectors' behavior in details and studied several design tradeoffs. Based on the evaluation, we propose several optimization techniques at different levels. Adaptive partial heap collection reduces the collection time dramatically without generational GC support. Memory prefetching techniques are studied as well. While hardware prefetching performs well with mark-compaction, pure software prefetching can reduce the collection time by up to 24%. We also found that good careful tuning of certain hotspots are more important than expected.

1. Introduction

Automatic garbage collection is popular in modern programming languages such as Java or C#. It relieves the burden of explicit memory management from programmer. Compared with copying collector, compacting collector does not reserve additional space for object moving, hence has better space efficiency. Moreover, it has no fragmentation problem that exists in mark-sweep collector. Compacting collector has another believed advantage in object locality by preserving the object order during collection. Because of the merits, compacting collectors are implemented in some commercial runtime systems [1][2]. Apache Harmony [3], an open source Java SE implementation, has two versions of compacting collectors built in. We call them GC-MC (mark&compact) and GC-CC (copy&compact) in this paper. The former is based on the classical Lisp2 algorithm [4], and the latter follows the well-known threaded algorithm originated from Jonkers [5] and Morris [6].

Compacting collector has also its downsides. A notorious issue is the long collection time. Since the compaction process moves objects on site, it can not finish in one pass. Besides the marking pass which identifies the live objects, a typical compaction algorithm needs additional pass(es) to compute object moving target and conduct the movement. Each pass usually means a full heap walking, which is time consuming and memory unfriendly. The other issue with compacting collector is its space overhead. Usually a compacting collector needs auxiliary space for mark-bit table, installing forwarding pointer (or offset table in recent work), and sometimes remembering reference slots as well.

Although various compaction algorithms were studied previously, their design tradeoffs and performance implications are yet to be understood in modern platforms with server workload. Recently the published work on compacting collectors are mainly about the parallelization and scalability, we think it is still important to understand the behavior and design tradeoffs in sequential compacting collectors, so as to provide a good foundation for their parallel version.

To achieve our goal, we evaluated two compacting collectors thoroughly in two different modern platforms with SPECJBB2005 benchmark. We compared the two collectors in details and developed several optimizations. The main contributions of this paper are:

1. We had apple-to-apple comparisons between two compacting collectors on real machines, giving detailed behavior characterizations and performance analyses;
2. We developed an adaptive partial heap collection mechanism that can reduce the compaction time dramatically while maximizing the overall GC throughput;
3. We studied memory prefetching techniques, and analyzed their applicability. Our software prefetching can improve SPECjbb2005 performance by up to 4.6%;
4. One key lesson we learned is, the actual performance of a collector is highly dependent on the underlying platform. A new memory hierarchy design can turn a collection algorithm from a loser to a winner.

We believe our work has laid a solid foundation for next step Harmony GC development. The framework also enables Harmony developers to implement and study other collection algorithms.

1.1 Apache Harmony GC Framework

Harmony is developed with modularity as one major pursuit. The GC component in Harmony interacts with the core runtime through a set of defined interface APIs. Any collector that implements the interface can be plugged into Harmony as a dynamically loaded object. At the time when this paper is written, there are three different garbage collectors implemented in Harmony, two of which are compacting collectors and are studied in this paper.

The paper is organized as follows. In Section 2 we briefly describe the compacting algorithms in Harmony, and then introduce our experimental platforms. In Section 3, we characterized the collectors in two platforms, a Unisys ES7000 and an Intel Tulsa system. We discuss adaptive partial heap collection in Section 4. In Section 5 we discuss the memory prefetching techniques. More tradeoffs during performance tuning are described in Section 6. Section 7 is related work discussion, and we conclude the work in Section 8.

2. The Collectors and Evaluation Platforms

In this section, we introduce the baseline algorithms of the two Harmony compacting collectors; then we describe our experimental platforms and evaluation methodology.

2.1 GC-MC compaction algorithm

Harmony has two compacting collectors implemented. They were developed based on two classical algorithms respectively.

GC-MC is based on the Lisp2 compacting algorithm [4]. The compaction process consists of four phases:

- *Marking phase* that traces from the roots set and marks all the live objects in the heap;
- *Repointing phase* that computes the new addresses of the live objects and install a forwarding pointer for each;
- *Fixing phase* that adjusts all the repointed references to point to their new locations;

- *Moving phase* that really moves the live objects to their new locations.

An explicit mark-stack is used during the marking phase. An object is marked by marking a bit in the mark-bit table, where one bit represents a four-byte word in the heap. The live objects are sliding-compacted to the lower end of the heap. Target addresses of all live objects are computed and written into the object headers as the forwarding pointers. In fixing phase, all the reference fields of live objects are updated to be the forwarding pointer of the referent object.

2.2 GC-CC compaction algorithm

The compaction algorithm in GC-CC is based on the threaded algorithm credited to Jonkers [5] and Morris [6]. It is more elaborate than GC-MC in that, it has only two phases (*marking* and *moving)* and does not need auxiliary data structure for forwarding pointer. GC-CC builds a reference list for every live object by reusing the original reference fields in the objects. The list for a live object links all the fields that contain references to the object. When an object is moved in moving phase, all references to it are updated by traversing the list.

References of two directions (pointing from low to high and from high to low) are treated differently when the list is built. Those references from high to low are added into the list in the marking phase, while the opposite direction references are added when their containing objects are moved.

2.3 Evaluation methodology

Since GC-MC and GC-CC share the same infrastructure in Harmony, we are able to have apple-to-apple comparisons between them.

We run Harmony in two different modern platforms. One is a Unisys ES7000 with eight 3.0GHz Intel Northwood processors and 400MHz FSB. The other is Intel Tulsa platform with four 3.2GHz Intel Pentium-D dual-core processors and 800MHz FSB. The Northwood processor each has 8KB L1 data cache, 512KB unified L2 on-chip cache, 4MB L3 unified cache and a 64-entry data TLB. A 32MB L4 cache is shared by 4 Northwood processors. One core of the Pentium-D processor has 16KB L1 data cache, 1MB unified L2 on-chip cache, 8MB L3 unified cache shared with another core in same processor and also a 64-entry data TLB. Tulsa platform is newer and has more advanced memory hierarchy.

SPECjbb2005 [7] is the workload we use. According to the reporting rule, a valid run of SPECjbb2005 in our platforms reports a final score based on the scores achieved from 8 through 16 warehouses. Our experiments showed that 1GB heap size means about 20% live object residency in the heap space with 8 warehouses, and about 40% with 16 warehouses. In this paper, we only show the data with 1GB heap size because we found different heap size does not seriously change our conclusions in GC module relative comparisons, although the absolute values and SPECjbb2005 scores are impacted. More workloads such as Dacapo [8] and SPECJAppServer [9] are also under investigation and we hope to report in future.

We use Intel Vtune [10], a performance tool, to uncover the detailed characteristics of the executions. Its event-based sampling (EBS) reads the hardware performance counters of the processor; hence we can get runtime information with minimum interference with the application execution.

3. Characterizations of GC-MC and GC-CC

In this section we characterize GC-MC and GC-CC in the Unisys and Tulsa platforms. To avoid the results being skewed by suboptimal implementations, the characterizations in this section include the optimizations that we will describe in following sections.

3.1 Characterizations in Unisys platform

We first give the average collection time breakdown of the compacting collectors. For GC-MC, the marking phase is further partitioned into the time for the first-time object touching (*mark first*), the time for remembering the re-pointed reference slots (*mark remember*) and the rest (*mark others*). Both the repointing and moving phases need to traverse the mark-bit table, so we separate the mark-bit table traversal time from their belonged phases: The time in repointing phase is *repoint traversal* and that in moving phase is *move traversal*. For GC-CC, we separate the list-related operation time into three parts: list building in marking phase (*mark list-build*), list updating in moving phase (*move list-update*), and reference updating in moving phase (*move list-reference*). Figure 1 shows the time breakdown of both collectors' average collection time over five runs in the Unisys platform.

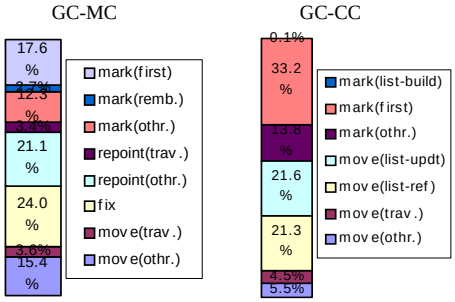

Figure 1. Execution time breakdown on Unisys

We can see from the figure that the four phases of GC-MC almost evenly partition the collection time, with a surprising exception that the moving phase takes the least time while marking takes most. This is surprising because people's intuition is the real object movement would be the most time consuming phase. In the marking phase, the first-time object touch takes a big portion (17.6%) that is almost equal to the moving phase time. This gives us an impression on how intensive the memory operations are in GC-MC. The situation is even more obvious in GC-CC, where the first-time object touch takes more than 30% of the total time. These imply good potentials for memory prefetching, which we will discuss later.

Table 1. Memory profile on Unisys

		8WH	10WH	12WH	14WH	16WH
IPC	GC-MC	0.087	0.087	0.088	0.090	0.090
	GC-CC	0.092	0.095	0.093	0.081	0.079
Loads per inst	GC-MC	0.451	0.432	0.443	0.443	0.445
	GC-CC	0.361	0.363	0.373	0.374	0.353
L1 miss ratio	GC-MC	8.41%	8.76%	8.31%	10.1%	9.05%
	GC-CC	5.75%	5.33%	5.61%	6.20%	5.62%
L2 miss ratio	GC-MC	3.62%	3.57%	4.02%	5.08%	3.74%
	GC-CC	2.54%	2.44%	2.42%	2.52%	2.70%
DTLB miss ratio	GC-MC	0.45%	0.48%	0.48%	0.48%	0.48%
	GC-CC	0.59%	0.63%	0.57%	0.68%	0.61%

Another interesting observation is that the list-related operations take more than 40% of GC-CC time. Because list is a dynamic data structure, it is not straightforward to improve its operation time. The list building part takes invisible percentage in the time bar, which means the number of references pointing from high to low is small. A side note is: This means a generational collector would match the behavior well.

Table 1 shows the memory access profile of the two collectors. We collected the IPC, number of loads per instruction, L1 cache miss ratio, L2 global cache miss ratio, and DTLB miss ratio. We can see GC-MC has more memory accesses, more cache misses but less DTLB misses. IPCs of both collectors are low in this platform. These might mean that both collectors' behavior does not match well with the underlying hardware, and the threaded reference list causes more page walks.

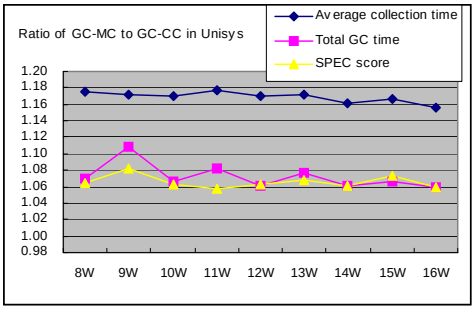

Figure 2. Performance comparison in Unisys

The overall performance comparisons between the two collectors are shown in Figure 2.

In Figure 2, the number is the lower is better. The average collection time of GC-MC is higher (~17%) than GC-CC, but the difference of total accumulated collection time during the whole application execution is not so big (~7%). That means GC-MC collections happen less times during a fixed period because of its longer time. The overall SPECJbb2005 score of GC-MC is worse than GC-CC because of its higher collection time. Note we use the reciprocal of the SPEC score for the comparison so that the proportional relation between GC time and SPEC score can be clearly demonstrated.

3.2 Characterization in the Tulsa platform

The function profile of both collectors in Tulsa platform is shown in Figure 3. Compared to Figure 1, the memory intensive operations take less percentage in Tulsa. For example, the fixing phase is 5% less in percentage and the first-time object touch is 10% less. The list manipulation part in GC-CC is reduced to 35%. This is because of the newer memory hierarchy design in Tulsa platform. The memory profile next confirms this assertion.

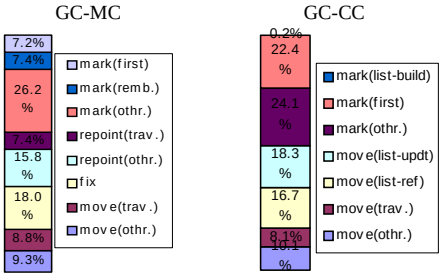

Figure 3. Execution time breakdown on Tulsa

Table 2 gives the memory profile in Tulsa platform. The IPCs of both collectors are increased by 4X~5X due to the reduced cache misses. The double-sized cache of Tulsa core can effectively reduce the cache miss ratios by 2X~3X. The table also shows that the cache access improvement in GC-CC is smaller compared to GC-MC, which is mainly due to the random memory access pattern of the reference list. We speculate the threaded algorithm would be less friendly to the latest progress in modern processor memory hierarchy.

The overall performance comparison in Tulsa is given in Figure 4. Interestingly, we find the curves of the two collectors reverse their positions compared to Figure 2: GC-MC performs uniformly better than GC-CC. This is not surprising though, since the intensive memory operations and their regular access pattern in GC-MC can benefit more from Tulsa. Figure 4 also shows proportional relation between total GC time and achieved SPEC score.

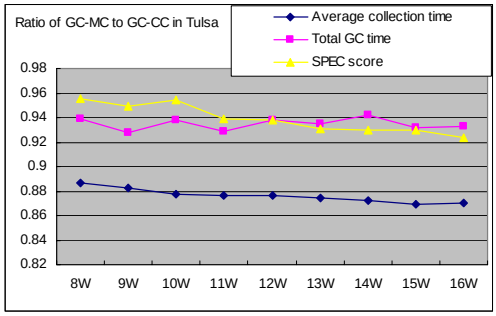

Figure 4. Performance comparison in Tulsa

Table 2. Memory profile on Tulsa

		8WH	10WH	12WH	14WH	16WH
IPC	GC-MC	0.409	0.408	0.408	0.428	0.398
	GC-CC	0.354	0.349	0.344	0.341	0.338
L1 miss ratio	GC-MC	3.66%	3.80%	3.92%	3.85%	3.99%
	GC-CC	3.06%	3.23%	3.33%	3.33%	3.18%
L2 miss ratio	GC-MC	1.31%	1.32%	1.40%	1.41%	1.38%
	GC-CC	0.95%	1.01%	1.11%	1.09%	1.11%
DTLB miss ratio	GC-MC	0.85%	0.84%	0.87%	0.88%	0.86%
	GC-CC	1.06%	1.09%	1.12%	1.13%	1.10%

4. Adaptive Partial Heap Collection

Because of the known long time of full heap compaction, incremental compaction was proposed [11][12]. The idea is to collect part of the heap normally and collect the whole heap only when necessary. We developed partial heap collection in both GC-MC and GC-CC. In this section, we first describe an adaptive mechanism we designed that determines when to trigger a full collection; then we discuss the application of a copying collector for partial collection.

4.1 The adaptive partial heap collection idea

We designed the partial heap collection by collecting only the objects allocated after last collection. This is actually similar to a generational GC except that we do not employ write barriers to remember the references from old objects to the newly allocated ones; instead, those references are discovered by scanning the heap. In this way the partial heap collection has the same marking phase as full heap collection. The computations in other phases will be much reduced. Since other phases in our collectors takes more than half of the total collection time in both platforms, we can expect an obvious GC time reduction.

The downside of partial collection is, since it reclaims only the newly allocated area, the space for those dead objects in old area can not be reclaimed. This will trigger more frequent collections, which are undesirable for overall application performance. In order to guarantee the effectiveness of partial collection, we propose an adaptive strategy to guide the collector to trigger partial or full collection. The goal of the strategy is to maximize the overall GC throughput, which is measured as the ratio of total produced free space in all the collections to the sum of all the collection times, i.e.,

$$Throughput = \Sigma SizeofFreeSpace \,/\, \Sigma TimeofCollection \quad (1)$$

We assume the free space size after a full collection is S_{max}, and the threshold free space size for triggering a full collection is S_{min}. After each partial collection, size of dS newly allocated objects can survive. This means for every $(S_{max} - S_{min})/dS$ times of partial collections, a full collection will be triggered. If each partial collection takes time T_{fast}, and a full collection spends

time T_{slow}, the total time spent for all the collections between two full collections is:

$$T = ((S_{max} - S_{min})/dS) * T_{fast} + T_{slow} \quad (2)$$

The total free space size produced during this period is (computed as a series):

$$S = (S_{max}+S_{min})*(S_{max} - S_{min} + dS)/2*dS \quad (3)$$

According to (1), the throughput approximation between two full collections is S/T. Since S_{max}, T_{slow}, T_{fast} and dS can be measured at runtime, the maximal S/T can be reached with a certain value of S_{min}. This S_{min} value is the threshold to trigger a full heap collection. When the remaining free space size is less than S_{min}, a full collection should be carried out.

We applied this adaptive strategy with GC-MC for its partial collection, and the average collection time is shown in Figure 5. The newly produced live data size dS is also shown.

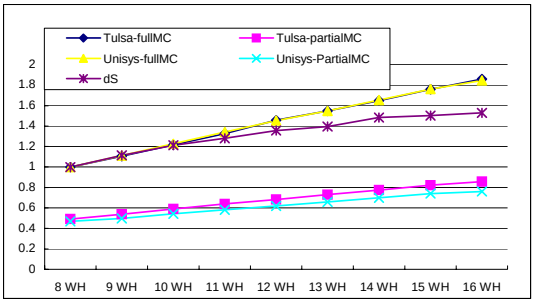

Figure 5. Normalized collection time and *dS*

The collection time is normalized to that of 8 warehouses. We can see that the partial collection time is less than half of the full collection time. And the partial collection time is roughly proportional to the newly produced live data size dS.

We measured the performance improvement of GC-MC with partial collection over full-collection-only GC-MC in both platforms. The results are shown in Figure 6. As a comparison, Figure 6 also shows the performance without the adaptive strategy. Without adaptation, *Smin* is half of the total heap size.

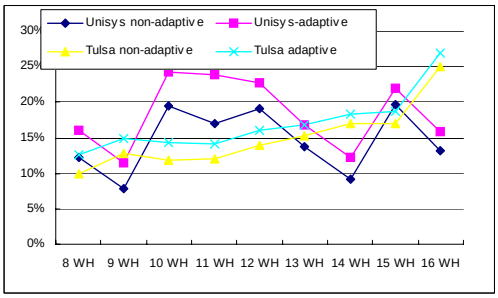

Figure 6. Performance speedups with partial heap collection with/without adaptive strategy

Figure 6 shows that the adaptive partial collection can improve the performance by more than 10% in both platforms, and the adaptive strategy itself contributes more than 2% in average.

An interesting observation is the partial collection can benefit more with more warehouses in Tulsa platform. As we discussed above, partial collection can not improve the marking phase, which becomes more and more dominant when the warehouse number increases, hence is benefited more.

4.2 Copying collector for partial heap collection

While the full collection uses compaction algorithm, we can employ a different one for the partial collection. Copying collector is a good candidate to reduce the partial collection time. The main reason for choosing copying collector is it has only one pass, and the condition for reserved copy space can be met normally. If the reserved space is not enough for copying, the collector can simply fall back to compaction algorithm. The copy reserve space can be much smaller than the collected space. The idea of using copying collection in common case and falling back to compaction is not new [13]; recently it is applied for mature space collection in a generational GC [14].

We implemented this design in GC-CC with a depth-first copying collector, and set the reserved space to be one-fifth of the from-space. We compared the average collection time of partial heap compaction and partial heap copying in two platforms. The results are shown in Figure 7. A copying collector is indeed faster: Its collection time is about 60% of partial compaction with 8 warehouses, and about 70% with 16 warehouses. In Tulsa, it saves less, because compaction can benefit more from its new memory hierarchy design.

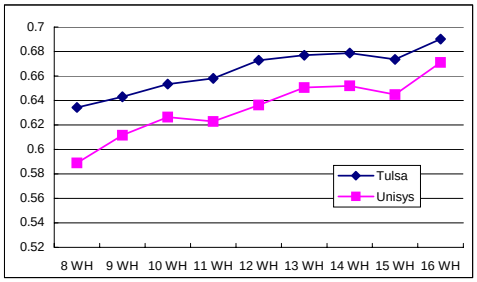

Figure 7. Ratio of average partial copying collection

We also collected the average collection time of full-heap GC-CC and partial-heap GC-CC with adaptive copying collector as shown in Figure 8. The partial copying collection time is less than one third of a full GC-CC collection.

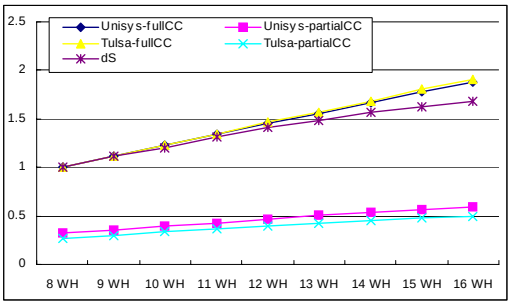

Figure 8. Normalized average collection time of full-heap and partial-heap GC-CC

However, faster partial heap collection does not necessarily mean overall better performance. In our experiments, GC-CC with partial copying actually has similar throughput as GC-MC with partial compaction. A believed disadvantage of copying collector is that, its object locality can be hurt because it does not preserve the object order. To verify this traditional wisdom, we collected the memory profile of mutator threads with both copying and compacting collectors, as shown in Figure 9.

The data confirm that copying collector causes more DTLB misses than the compacting collector in both Unisys and Tulsa platforms. Nevertheless, we find the L1 cache miss rates of them are very close. This means that object order preservation is more important to the page-level locality than the cache-level locality for SPECjbb2005. We think the reason is that the depth-first copying collector can well keep the cache-level locality, but can not with the page-level locality. In our experiments, the DTLB misses can slow down the mutator execution by up to 4%. The TLB misses can be reduced with larger hardware page size, while that's true for both collectors.

5. Memory Prefetching Techniques

As we discussed in previous sections, the intensive memory operations in the collectors may imply the importance of a good memory prefetcher. In this section, we discuss the

prefetching techniques used in our compacting collectors. We address the topic in hardware prefetching and software prefetching separately.

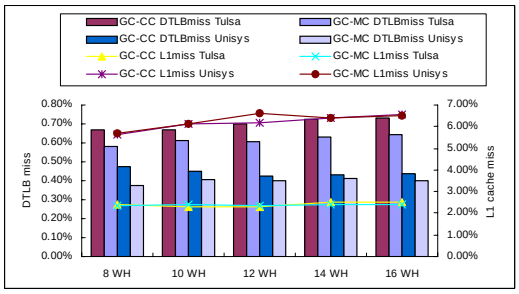

Figure 9. Memory profile of the Mutator

5.1 Harware prefetching

Tulsa has two hardware prefetchers [15] which can be turned on/off via BIOS setting. They are *adjacent cache line prefetcher* that prefetches the adjacent 64-byte line in a 128bytes sector when a cache line is loaded, and *stride prefetcher* that attempts to stay 256 bytes ahead to prefetch the cache line.

We measured the collection time of different collector configurations (full heap GC-MC, full heap GC-CC, partial heap compaction, and partial heap copying) with Tulsa hardware prefetchers. The speedups are given in Table 3.

Table 3. Speedups with hardware prefethers

Collectors / Prefetchers	Full GC-MC	Full GC-CC	Partial compaction	Partial copying
Stride	2.00X	1.53X	1.53X	1.88X
Adjacent	1.42X	1.40X	1.61X	1.54X
Both on	2.47X	1.82X	2.12X	2.25X

The speedups are significant for all the collectors, while different prefetchers bring different improvements. Stride prefetcher can benefit *full GC-MC* and *partial copying* more than *full GC-CC* and *partial compaction*. For *full GC-CC*, it is because it uses threaded algorithm so that the list access pattern is rather random. For *partial compaction*, its marking phase dominates the time and the object graph tracing order for marking is also rather random. On the other hand, *partial compaction* can benefit most from adjacent prefetcher. This is because the scanning of live objects often accesses adjacent area of the object. Except the *full GC-CC* compaction, all the collectors achieve more than 2X performance when both prefetchers are turned on.

5.2 Software prefeteching

We implemented three different pure software prefetching algorithms in all the collectors. We used them to improve marking phase, since it is most time consuming and most memory intensive.

- *prefetch-on-grey(POG)*[16]: The collector prefetches target object when it is pushed onto the mark stack, which has a prefetch distance equal to the interval between the time an object is pushed and the time it is popped.
- *buffered-prefetch(BP)* [17]: The collector maintains an extra prefetch buffer queue. Objects are enqueued to the buffer from the mark stack till the queue is full or the mark stack is empty. An object in tail of the buffer is prefetched while the object in head of the buffer is scanned.
- *prefetch-without-mark(PWM)*[18]: The collector puts all reference fields onto the mark stack instead of only those unmarked objects, so as to delay the access to the objects. A referenced object can be prefetched right before it is going to be checked and marked.

We tested the three prefetching techniques. Table 4 lists the biggest performance improvement for different collection configurations and their respective prefetchers.

Table 4. Benefit of software prefetching

	Full GC-MC	Full GC-CC	Partial compaction	Partial Copying
Prefetcher	POG	POG	POG	PWM
Speedup	1.11X	1.00X	1.24X	1.21X

Partial compaction benefits mostly from *POG* prefetcher, because of its big portion of marking time. *Full GC-MC*'s speedup is less than half of *partial compaction*'s. On the other hand, the threaded algorithm (*full GC-CC*) can not benefit from any software prefetcher, because it needs to access the objects to build the list, hence no prefetch distance. *Partial copying* can benefit most from the PWM prefetcher even with the overhead caused by pushing more objects onto mark stack. Although *buffered-prefetch* can not really bring performance speedup due to its overhead, we did observe obvious cache miss reduction.

5.3 Hybrid prefeteching

Performance with both hardware and software prefetchers (*hybrid prefetching*) is given in Table 5. All are better than or equal to the product of the speedups achieved by the two prefetchers when applied separately except *partial copying*. *Partial copying* cannot get the double pay as others possibly due to the high bandwidth requirement of the *PWM* strategy.

Table 5 Benefits of hybrid prefetching

Full GC-MC	Full GC-CC	Partial compaction	Partial Copying
2.74X	1.82X	2.67X	2.54X

When the hardware prefetchers are on and GC-MC with partial collection is used, software prefetching (*POG*) can further reduce total GC time of SPECjbb2005 execution by 18% in Tulsa. Consequently, SPECjbb2005 performance has 4.6% further improvement compared to the hardware prefetching baseline.

6. Other Design Tradeoffs

There are other design tradeoffs worth mentioning which impact the final delivered performance.

6.1 Remember-set vs. heap scanning

One tradeoff is to use extra space for remember-set to keep repointed reference slots found in during marking, rather than scanning the heap a second pass to find them for fixing the pointers. This is to trade space for heap scanning time. Although Abuaiadh et al. [19] think it is unacceptable to remember all the reference slots for full heap compaction, our experiments suggest the real space overhead is less than 5% even for 16 warehouses. Table 6 lists the impact on space and collection time when remember-set is not used. It shows the collection time can be about 9% more if without remember-set. In our experiments, the overall SPEC score is reduced by 2.3%.

Table 6. Effect of removing the remember-set

Partial compaction				Full GC-MC			
8WH		16WH		8WH		16WH	
Space	Time	Space	Time	Space	Time	Space	Time
-0.2%	9.4%	-0.4%	9.9%	-2.3%	7.4%	-4.5%	7.1%

6.2 Mark-bit table traversal vs. object access

Another design tradeoff is with the mark-bit table traversal. People usually believe that the mark-bit table traversal is much cheaper than the heap traversal because of its much smaller size (usually 1/32 of the heap size). Some researchers even proposed to mark the bit for the last word of an object [20], so that the size of an object can be got by traversing the mark-bit table without accessing the object header metadata.

Bit manipulation can cause unexpected overhead if used improperly. We found that our "bit-skip" approach improves the mark-bit table traversal performance. This approach skips

the bits for current object body by computing its size with information in object header, and then checks the following bits to identify next live object location. We also employed the fast bit scanning instruction "BSF" [21] in Intel X86 processors to further reduce the bits operations. Table 7 gives the speedups of the given phases with the two optimizations in Tulsa platform.

Table 7. Speedups by mark-bit table traversal optimizations

	Repointing (GC-MC)	Moving (GC-MC)	Moving (GC-CC)
bit-skip	1.24X	1.09X	1.22X
bit-skip+ BSF	1.37X	1.43X	1.25X

6.3 Space zeroing site and size

One another very interesting design tradeoff is the invocation site for "space zeroing", i.e., to nullify the space before it is used for object allocation. We found the most effective way is to zero the space before object allocation but with carefully tuned space zeroing size (2KB in our case). Basically, the zeroing actually acts as software prefetching. We do not want to zero the space too early such as right after a collection, and we need a suitable size to amortize the zeroing overhead.

7. Related Work

Blackburn et al. [22] gave a comprehensive study on the performance impact of various GC strategies. Different applications are characterized with different heap sizes, and different GC algorithms. Our paper is focused on compacting collectors.

An early work [23] on performance comparison of compacting collectors gave algorithm complexity analysis on four different compacting collectors and summarized that the Lisp 2 algorithm performs best. The four collectors' detailed descriptions can be found in [4][5][6][24] respectively. Our work demonstrates that algorithm complexity differs from real performance.

The idea of partial heap collection is partly inspired by incremental compaction [11][12]. We implemented and evaluated two different collection algorithms and partial copying collector performs better than partial compacting collector. Our adaptive strategy is inspired by dual-mode GC [13] which switched between two modes of collections adaptively by computing the residency. In our work, the switch is triggered by a dynamically computed space size threshold.

Boehm [16] presented the first work that uses software prefetching to reduce cache misses. It issues prefetching for a live object when it is marked and put onto the mark stack. Cher et al.[17] argued that this strategy suffers from the variable prefetch distance which may cause the fetched data be replaced from the cache before it is really used. They proposed a buffered prefetch approach and showed better performance with simulation. Our experiments found the overhead of buffered prefetch is too high to be really beneficial.

Diwan et al. [25] studied the memory subsystem performance using memory simulator. Huang et al. [26] studied cache access locality improvement with object online reordering technique. Abuaiadh et al. [19] shows order preserving during collection is important for object locality. Our experiments in real machines showed that, by preserving allocation order, the TLB locality of compacting collector is better than copying collector.

Recently the research about compacting collectors are more focused on the parallel/concurrent design [19][20], and Abuaiadh et al. developed a new compacting collector that even the restricted parallel version can outperform the threaded algorithm.

We believe what we learned in this paper are helpful for the parallel/concurrent GC developers to select their baseline sequential compaction algorithm. A parallel generational compacting collector is recently developed for Apache Harmony based on GC-MC.

8. Summary

In this work, we have extensive evaluations on two compacting collectors in Apache Harmony, and studied several design tradeoffs. The adaptive mechanism and the memory prefetching techniques we developed were proved effective to improve the compacting collectors' performance.

We found the best performing compacting collector algorithm highly depends on the target platform, the applied optimizations and performance tunings. The actual result can be contrary to the algorithmic complexity analysis. The processor memory hierarchy design sometimes decides the final winner. Our experiments suggest that a combination of GC-MC full heap compaction with partial copying collection can perform best with SPECJbb2005 in Tulsa platform.

Our next step is to apply what we learned in this work to more advanced collector design with more workloads study.

References

[1] S. Borman. S. Sanitation, Understanding the IBM Java Garbage Collector, http://www.ibm.com/.

[2] N. Nagarajayya and J. Steven Mayer, Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1. http://developers.sun.com/

[3] Apache Harmony, http://incubator.apache.org/harmony.

[4] R. E. Jones. Garbage Collection: Algorithm for Automatic Dynamic Memory Management. Wiley, Chichester, July 1996.

[5] H.B. M. Jonkers. A fast garbage compaction algorithm. Information Processing Letters, July 1979

[6] F. L. Morris. A time- and space-efficient garbage compaction algorithm. Communications of the ACM, 21(8):662-5 1978

[7] http://www.spec.org/jbb2005/

[8] The Dacapo benchmark suite. http://dacapobench.org/

[9] http://www.spec.org/jAppServer2004/

[10] Vtune Performance Analyzer. http://www.intel.com/cd/software

[11] B. Lang and F. Duport. Incremental incrementally compacting garbage collection. In SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, volume 22(7) of ACM

[12] O.B.Yitzhak, I. Goft, E. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In David Detlefs, editor, ISMM'02.

[13] P. Sansom. Dual-Mode Garbage Collection. In Third Int. Workshop on the Parallel Implementation of Functional Languages, September 1991."

[14] P. McGachey and A. L. Hosking, Reducing Generational Copy Reserve Overhead with Fallback Compaction, ISMM'06

[15] Intel Corporation. IA32 Intel® Architecture Optimization Reference Manual.

[16] Boehm, H.-J. Reducing garbage collector cache misses. ISMM'00.

[17] C. Y. Cher, A. L. Hosking, T. N. Vijaykumar, Software Prefetching for Mark-Sweep Garbage Collection: Hardware Analysis and Software Redesign. ASPLOS'04.

[18] R. Garner, S. Blackburn, D. Frampton, Effective prefetch for mark sweep garbage collection, Talk given in Intel, Oct. 2006

[19] D. Abuaiadh, Y. Ossia, E. Petrank and U. Silbershtein, An efficient parallel heap compaction algorithm. OOPSLA'04.

[20] H. Kermany, E. Petrank, The compressor: Concurrent, Incremental, and parallel compaction, PLDI'06.

[21] Intel Corp. Intel Architecture Software Developer's manual.

[22] S. M. Blackburn, P. Cheng and K. S. Mckinley, Myths and Realities: The Performance Impact of Garbage Collection. SIGMETRICS'04

[23] J. Cohen and A. Nicolau, Comparison of compacting algorithms for garbage collection. ACM Transactions on Programming Languages and Systems, 5(4):532-553, 1983

[24] Wilson. P. R. Uniprocessor Garbage Collection Techniques. Technical Report. University of Texas, Janualry, 1995

[25] A. Diwan, D. Tarditi and E. Moss, Memory Subsystem Performance of Programs Using Copying Garbage Collection. POPL'94.

[26]X.Huang, S. M. Blackburn, K. S. Mckinley, J. B. Moss. The Garbage Collection Advantage: Improving Program Locality, In OOPSLA'04