Mostly-Copying Collection: A Viable Alternative to Conservative Mark-Sweep *

Frederick Smith† fms@cs.cornell.edu

Greg Morrisett jgm@cs.cornell.edu

Computer Science Department
Cornell University
Ithaca, New York 14853-7501, USA

Abstract. Many high-level language compilers generate C code and then invoke a C compiler to do code generation, register allocation, stack management, and low-level optimization. To date, most of these compilers link the resulting code against a conservative mark-sweep garbage collector in order to reclaim unused memory. We introduce a new collector, MCC, based on *mostly-copying collection*, and characterize the conditions that favor such a collector over a mark-sweep collector. In particular we demonstrate that mostly-copying collection outperforms conservative mark-sweep under the same conditions that accurate copying collection outperforms accurate mark-sweep: Specifically, MCC meets or exceeds the performance of a mature mark-sweep collector when allocation rates are high, and physical memory is large relative to the live data.

1 High Level Overview

Languages such as C [1] and C++ [27] are *GC-unfriendly* because they allow programs to violate the key premise of tracing garbage collection—that all live objects are reachable by a sequence of pointer traversals from the root set. For example, a program might violate this premise by using pointer arithmetic or casting. Consequently, compilers for GC-unfriendly languages do not support garbage collection and do not provide the information necessary to distinguish pointers from other data. This makes accurate garbage collection infeasible.

Conservative tracing garbage collection is a practical memory management approach for GC-unfriendly languages [14, 11, 13, 12, 16, 8, 7, 9, 10]. This approach is *practical* because only minor modifications are necessary to add memory management to most programs; and no special compiler support is required.

* This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and ARPA/RADC grant F30602-96-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

† Supported by a National Science Foundation Graduate Fellowship.

‡ Instead, a conservative garbage collector relies on the programmer, rather than the compiler, to ensure that the program does not violate the invariants it requires. For a typical collector these invariants might include not mangling pointers, always retaining a pointer to the beginning of an object that is in use, and only using the collector's allocation routines.

Indeed, conservative collectors are so practical that a variety of compilers for GC-friendly languages, including TIL/C [29, 20, 28, 30], Toba [22], Harissa [21], Bigloo [24] and Vortex [15], have chosen to use a conservative collector and to compile to C rather than use an accurate collector and generate assembly code. The implementors of these systems have decided that the portability of C, and the optimizations afforded by pre-existing C compilers outweigh the benefits of accurate collection.

Given the many compilers using conservative collection it is remarkable that all of the above compilers link the resulting code against the same conservative collector: The Boehm-Weiser collector (BWC) [14]. BWC is based on a mark-sweep algorithm, and was designed to be used in systems where type information is generally unavailable. Compared to copying collectors, mark-sweep algorithms tend to require less memory, but have greater allocation costs, the potential for greater fragmentation, and a running time that is proportional to the size of the heap.

In this paper, we present a new collector based on an improved version of the mostly-copying collection algorithm of Bartlett [7, 8, 10] as an alternative to BWC. Our collector (MCC), in contrast to BWC, is targeted at code generated by compilers, and thus expects type information to be available for most, but not all objects. Because MCC is mostly a copying collector it has many of the same performance characteristics as an accurate copying collector: compaction, fast allocation, and running time proportional to the live data [19].

Briefly this is how MCC achieves these characteristics. To achieve fast allocation, MCC organizes the heap into *pages*. The client first allocates a page from the collector, and then fills the page with objects in a contiguous fashion. Thus, allocation has the same amortized cost as for an accurate copying collector. To achieve compaction and running time proportional to the live data, MCC, when invoked, attempts to perform a standard copying collection. However, when type information is unavailable for a word that appears to be a reference, the collector marks the referenced object and *pins* the page containing the object. Objects on pinned pages remain in place, but all other objects are copied and compacted onto free pages. After traversal, the collector reclaims all unused pages.

To validate the claim that mostly-copying collection is a viable alternative to conservative mark-sweep we compared the performance of eight ML benchmarks using both MCC and BWC for a modified version of the TIL compiler that generates C code, TIL/C. The results show that the characteristics of each program determine which collector performs best, and that the tradeoffs involved in choosing MCC over BWC are the same as those involved in choosing an accurate copying collector over an accurate mark-sweep collector. In particular, MCC

‡ See [13] for a discussion of what compiler support is required.

is favored when physical memory is plentiful, when the amount of live data is small relative to the amount of allocation, and when allocation rates are high. In contrast, BWC is favored when memory is scarce, and when the live data set is large. We found that the overall performance of some benchmarks changed by as much as 40% depending on the collector.

The main results of this paper are:

- A new mostly-copying collection algorithm that achieves fast allocation, and allows for untyped objects to co-exist in the heap and to be collected.
- A new mostly-copying collector called MCC.
- A comparison between BWC and MCC in the context of an ML to C compiler.

The rest of the paper is organized as follows. Section 2 discusses the algorithms employed in BWC to achieve good performance in a conservative environment. Section 3 introduces our new mostly-copying collection algorithm and the implementation of MCC. Section 4 provides data comparing MCC to BWC for eight ML benchmarks that showcase different aspects of the two collectors. This paper requires familiarity with many garbage collection concepts that for the sake of brevity have been omitted. The reader requiring more clarification is referred to Wilson's survey on garbage collection techniques [32] or Richard Jones' text [19].

2 The Boehm-Weiser Conservative Collector

BWC [14, 11] is a conservative mark-sweep collector available on a wide variety of platforms. It is used by at least ten systems to date and has been under development since 1987 [§]. BWC is very successful because it is easy to use (programs need only redirect `malloc` and `free` to the appropriate collector routines) and has performance comparable to explicit memory management for almost all programs [33]. In the remainder of this section, we briefly describe a simplified version of the algorithm and data structures that BWC employs.

BWC breaks the heap into 4 KB blocks. Each block contains objects of a particular size, and has a header associated with it. The *header* records the size of the objects allocated in the block, and a bitmask indicating the objects that have been marked. Blocks are requested from the operating system through the standard `malloc` as needed, and freed blocks are maintained on a free list. Separate free lists are maintained for each object size.

A client allocates objects by calling either `GC_malloc` or `GC_malloc_atomic`. The former is used to allocate arbitrary data, and the latter is used to allocate pointer free objects (*atomic* objects). Although no objects have to be allocated using `GC_malloc_atomic`, doing so improves BWC's performance.

The collector also provides macros for inlining allocation for improved performance in allocation-intensive code. On a Sparc, the inlined allocation code

[§] See `http://reality.sgi.com/employees/boehm_mti/gc.html`.

for the fast path takes 9 instructions: 3 *loads*, 3 *stores*, 1 *branch*, 1 *compare*, and 1 *add*. Most of the memory operations are related to managing the free list. 1 *store*, 1 *load*, and 1 *add* are used to track the number of objects allocated of a given size and could potentially be eliminated. Unfortunately, dependencies within the collector make this change difficult to make.

When a collection starts, the collector scans the *root set* (*i.e.*, the stack, static area, and registers) for references to objects. Each reference that is found is pushed onto a mark stack, and the associated object is marked. The collector then performs the following steps until the mark stack is empty. The top object reference is popped off the stack and each word of the object is tested to determine whether it is a pointer to an unmarked object (how this is done is discussed in the next paragraph). If so, the pointer is pushed onto the mark stack and the object to which it refers is marked. Once the mark stack is empty, the collector sequentially sweeps a few blocks to free objects that are not marked, and returns control to the client. At each subsequent call to `GC_malloc`, the collector sweeps some more blocks until the whole heap has been swept. This *lazy-sweeping* reduces collector latency.

BWC uses a series of tests to determine whether a value in the heap is a pointer:

1. Is the value aligned as required by the architecture?
2. Does the value lie between the least and the greatest valid heap addresses?
3. Does the collector own the block the value references?
4. Does the value point to the beginning of an object?
   This test makes use of the fact that objects are segregated based on size. It tests whether the value points to an offset from the beginning of the block that is a multiple of the size of the objects on that block.

Almost all false references fail the first two tests, taking only a few instructions. A successful test for a pointer will take approximately 30 instructions [19]. The speed of pointer testing makes completely conservative collectors such as BWC possible.

3 MCC Algorithm & Implementation

In this section, we present the details of our mostly copying collector MCC, which extends the algorithm presented by Bartlett [9, 8, 10]. Section 3.1 discusses object representations and allocation. Section 3.2 presents the overall structure of the heap and the data structures used during collection. Section 3.3 presents our algorithm for determining whether a value in the heap is a pointer. Section 3.4 discusses the collection algorithm itself. Finally, Section 3.5 summarizes the key differences between our algorithm and other related work.

To simplify the presentation, we do not cover secondary issues such as large objects, alignment restrictions, determining stack and static area boundaries, etc., although these details are handled in the implementation.

3.1 Objects and Allocation

For our implementation of MCC, every object has a header word prepended to it. The header word determines the size, type, and state of the associated object. The collector recognizes three types of objects: record, array, and conservative. For records the header word also contains a bitmask that determines accurately which words of the object are pointers. It is this type information that allows MCC to copy objects and update pointers. During a collection, two bits in the header word record the state of each object:

- *Normal:* The object is unprocessed.
- *Forwarded:* The object has been copied and the header word has been replaced with a forwarding pointer.
- *Pinned:* The object cannot be moved because a conservative object might refer to it.
- *Marked:* The object was pinned, but has been found to be live.

A client can allocate either accurate objects or conservative objects. As in BWC, conservative objects require no type information and are allocated via the routines `gcMalloc` or `gcMallocAtomic`.

To allocate accurate objects, the client maintains a pointer `gcAllocPtr` into a free page (Pages are discussed next). If there is enough space for the object on the page, then `gcAllocPtr` is incremented by the size of the object. Otherwise, a new page is requested – at this point MCC may choose to invoke a collection. As a result, allocation for accurate objects is fast compared to BWC. In particular, once the client has a page, the client can allocate without interruption by the collector until the page is full. For a typical page size of 2 KB, the client can allocate about 150 cons-cells (list elements of 2 words + 1 header word) between interruptions. By keeping `gcAllocPtr` in a global register, the fast path instruction sequence for allocating an accurate object is 7 instructions: 1 *store*, 2 *adds*, 1 *and*, 1 *compare*, 1 *branch*, and 1 *move*. The *store* is for the header word. Of these instructions 1 *and*, 1 *add*, 1 *compare*, and 1 *branch* are used to determine whether there is enough space to proceed. With additional changes to the compiler we could achieve an amortized performance of 3 instructions per object allocation using the same techniques suggested by Appel [3].

3.2 Heap Structure

MCC maintains a heap structure consisting of heap blocks which are divided into pages. The collector allocates large (1/2 MB) heap blocks using `memalign` so that the most significant bits uniquely identify the block. The collector uses these bits to track which heap blocks it has acquired.

Each heap block is divided into pages (2 KB). Associated with each page is a *pageInfo* structure consisting of four words. Three of the words are used for miscellaneous list maintenance. One word serves as a header word for the page, and denotes its status: free, system, user, or pinned. Free pages are not in use by the client or the collector. System pages are used to maintain the pageInfo structures and various lists that the algorithm uses. User pages contain objects allocated by the client. Pinned pages are user pages that contain at least one live object that cannot be moved (*i.e.*, one marked object). No pages are pinned except during a collection.

MCC maintains the following data structures:

- `free list`: A list of pages that are not currently in use.
- `from space`: A list of newly filled user pages, and pages that survived prior collections.
- `cheney queue`: A standard Cheney queue like that used in any copying collector. In MCC the queue is implemented as a list of pages with pointers to the beginning and to the end of the queue — not as a contiguous block of memory.
- `mark queue`: A queue of references to marked objects whose children have not been processed yet. This is analogous to the mark stack used in a mark-sweep collector.
- `pinned page list`: A list of all the pages containing marked objects.
- `conservative object list`: A list of conservative objects that were alive after the last collection or have been allocated since then.

3.3 Pointer Testing

MCC implements a simple pointer testing scheme similar to the one used in BWC. It performs the following tests on a value to determine whether it is a pointer:

1. Is it aligned for the architecture?
2. Is it between the least and the greatest heap addresses?
3. Does it reference a valid heap block?
4. Does it reference a user page?
5. Does it reference the beginning of an object?
   This test starts at the beginning of the page and scans from one header word to the next until the appropriate object is found. Consequently, this algorithm has an average running time proportional to one-half the number of objects on a page.

Most false references fail the first 3 tests. Successful pointer testing is more expensive for MCC than for BWC because of test 5. Because BWC segregates objects based on size they are able to speed this up considerably. MCC does not have that advantage and so must use a slower test or additional data structures.

3.4 Collection Algorithm

Later we give a detailed list explaining our collection algorithm in four steps; however, intuitively our algorithm can be understood as a hybrid conservative mark-sweep and copying collector. The first two steps (Pinning and Root Marking) determine the set of objects that will not be moved in the subsequent steps, and fill the `mark queue` with the roots. Graph traversal then processes objects off two parallel queues – the `cheney queue`, which contains objects that have been copied, and the `mark queue`, which contains objects that cannot be moved – until no objects remain. The final step (Sweeping) sweeps the pinned pages (in other words, pages containing marked objects) to unmark the objects on them. Lastly all the pages that are not pinned, or not in the `cheney queue` are freed.

Note that our algorithm heavily favors copying over mark-sweep. In particular, a marked object is touched four times during a collection: once to pin it, once to mark it, once to scan it, and once to unmark it. In contrast copied objects are touched only twice: once to copy the object, and once to scan it.

In detail, a collection proceeds as follows:

1. *Pinning:* For each object in the `conservative object list`, pin its children. This step guarantees that the collector will not inadvertently move an object in the following steps. However, some objects may be pinned unnecessarily.
2. *Root Marking:* Scan the root set. Mark all the objects referenced by it. Enqueue the objects on the `mark queue`, and pin their pages.
3. *Graph Traversal:* Traverse the graph by arbitrarily dequeuing objects from either the `mark queue` or the `cheney queue` until both queues are empty. For each child of the dequeued object, if the child is:
   - *pinned,* mark it, enqueue it on the `mark queue,` and pin its page ¶.
   - *marked,* do nothing.
   - *forwarded,* update the pointer to the object's new location.
   - *normal,* enqueue it on the `cheney queue` (*i.e.*, copy it), install a forwarding pointer, and update the parent's pointer.
4. *Sweeping:* Traverse the `pinned page list` unmarking the marked objects on each page and erasing the remaining objects to avoid future false references. Then append all the previously pinned pages to the `cheney queue`. Free all the pages in `from space` by adding them to the `free list`. Finally, make the `cheney queue` the new `from space`.

3.5 Comparison to Related Work

Our algorithm improves Bartlett's algorithm in various ways: it allows arbitrary conservative objects in the heap, it collects conservative objects, and it only performs one pass over the live data. In addition, MCC supports fast allocation for all objects, has less false retention, and produces less fragmentation. These improvements come at the cost of requiring two bits of state with each object, and slower processing of conservative objects.

The key data structure that allows MCC to avoid doing two passes over the live data is a list of all the conservative objects that might pin children. Using this list, MCC conservatively approximates the objects that it is not safe to

¶ Whenever we pin a page for the first time we remove it from `from space` and append it the `pinned page list`.

move. In contrast, Bartlett's algorithm does not maintain this list and therefore must do a second pass over the live data to unroll copies that should not have been made. The advantage of Bartlett's approach is that no state needs to be maintained with each object.

MCC achieves less false retention because it only retains objects that have been marked whereas Bartlett's algorithm retains all the objects on a page containing any marked object. Bartlett's algorithm does this because it uses a Big Bag of Pages (BiBoP) approach (only objects of the same type are stored on a given page). BiBoP obviates the need for a header word per object but it also makes it difficult to associate state with each object, and to allocate quickly.

To the best of our knowledge since Bartlett's original work there has only been one other mostly-copying collector: CMM [4, 5, 6] (for C++). CMM improves on Bartlett's algorithm in that it associates a bitmap with each page which allows it to mark individual objects, thus achieving less false retention. However, CMM did not make any fundamental changes and therefore still requires two passes over the live data. Some data comparing CMM to Bartlett's collector and to BWC in the context of C++ is published in [4].

4 Comparison of Overall Costs for ML

To compare MCC to BWC, we added a C back end to the TIL compiler [30, 29] and targeted it to use both collectors. We chose TIL because we were familiar with it, and because we were able to generate fairly natural C code. In particular, the generated code uses the C stack, resorts to almost no inline assembly (only enough for tail call elimination), and produces natural C constructs such as while loops.

We compiled eight small to medium sized benchmarks for ML, using BWC version 4.10 and then MCC. The code, except for allocation and collection, is the same for both versions (note, no header words are generated for BWC). We would have liked to compile larger, longer running ML programs, but most such programs use the ML-module system which TIL/C does not currently support. Instead we selected benchmarks that have been used in the literature [20, 28, 26, 25] to measure the performance of TIL and SML/NJ [2]. Since most of these ran very quickly, we modified the programs slightly to make them run longer (*e.g.*, by increasing the data set sizes or number of iterations). We compiled the C code using `gcc -O2` on a 256 MB Sparc 20 running Solaris 2.5.

To improve BWC's performance we inlined almost all allocations, allocated pointer free objects atomically, configured BWC for large heaps, and disabled interior pointers. We would have further liked to improve BWC's code for allocation, and to turn off lazy-sweeping. Lazy-sweeping improves latency for the collector but may hurt overall running time. We were not able to make these two changes without significant rewriting.

To improve MCC's performance we used `gcc` extensions to put the allocation pointer in a global register, inlined almost all allocations, and provided type information for all objects whose types were statically known. The only allocations we did not inline were for untyped objects. In practice there was only one benchmark (Groebner) where type information was not known for all the objects in the heap.

Once we had generated the code, we instrumented both collectors to time overall running times (wall clock, system and user times), and collection times. We collected timing information for a variety of heap sizes ranging from 1 MB up to 64 MB, for all of the above variables. Due to the mark stack, we gave BWC 1/2 MB more space than MCC. This extra space was only used for the mark stack and other internal data structures — not for allocation. However, this put MCC at a slight disadvantage, especially for the smaller heap sizes.

The actual numbers for all the experiments are available in Appendix A, as well as a detailed discussion of our methodology. In the next section we examine each benchmark individually to explain its performance. The end of this section summarizes the findings of that analysis.

Condition	Favors	
	BWC	MCC
Deep stack	x	
Limited type information	x	
Limited memory	x	
Small live data set		x
Intensive allocation		x
Varied object sizes		?

Table 1. Conditions under which BWC or MCC are favored.

Our analysis indicates that there are particular factors that determine which collector will perform better; however, for almost all cases the two collectors are comparable. Table 1 indicates conditions under which either MCC or BWC are favored.

When not much type information is available BWC is favored. For example, this is the case when the stack is deep or when there are many conservative objects in the heap. When a value is not a pointer, both BWC and MCC are comparable since most false references fail during the first tests. However, for a successful pointer test, BWC's algorithm takes about 30 instructions to determine that a particular value is a pointer, whereas MCC's algorithm takes on the order of 200 instructions. MCC does not implement a smarter algorithm because it does many fewer pointer tests than BWC. The results in the next section suggest that this decision may be worth re-exploring.

BWC requires less memory for two reasons: it does not copy objects, and it does not require header words on each objects. The former observation means that MCC needs at least enough additional space to fit the live data. In principle this could be as much as half the heap. The latter observation means that for languages, such as ML, where most objects are cons-cells (2 word objects) there is an additional space requirement of 50% for MCC. When these two facts are combined they suggest that MCC may require as much as three times the space that BWC needs. Despite the additional space overhead, MCC still has to do about twice as many collections as BWC.

Two facts mitigate BWC's space advantage. First, BWC double word aligns all objects that are greater than two words in size. So for objects of odd size, BWC loses the advantages of not having a header word. Second, other languages such as Java [17] and C++ require a header word (*i.e.*, a class pointer). In those contexts the header word would not be a factor.

MCC is favored by small amounts of live data, because the copy is cheap, and no work proportional to the heap is necessary. In addition, these cases typically have the highest allocation rates, thus enhancing the effect of MCC's cheaper allocation routines.

MCC may be at an advantage when objects of varied sizes are present because BWC maintains separate free lists for each size. Maintaining multiple lists may slow down both allocation and collection. Unfortunately, the magnitude of this effect is not evident from the available data.

Most of the benchmarks show that MCC and BWC are comparable – within 5%. For benchmarks with very small amounts of live data MCC may do as much as 40% better than BWC. However, when the stack is deep or many conservative objects are residing in the heap, MCC may be around 25% slower. Overall the analysis suggests that mostly-copying collection is a competitive technique.

4.1 Benchmarks

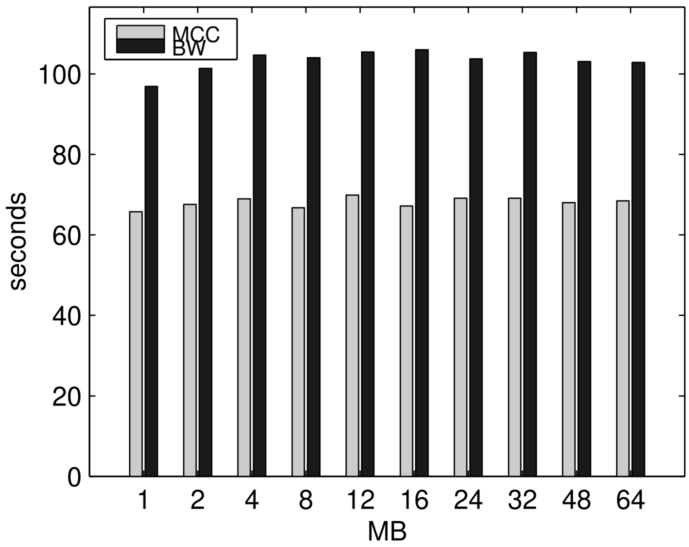

Fig. 1. Checksum: Wall clock times as a function of heap size.

Checksum This benchmarks performs a checksum on a stream of 16-bit values. The state of the computation and the stream require the allocation of many 2 and 3 word objects.

Checksum allocates almost 1 GB of data,, but on average there are only 46 live objects. This is an extreme example of the advantages of copying collection. Since performance is proportional to the live data rather than the heap and MCC has faster allocation than BWC, MCC runs this benchmark in approximately 65 seconds as opposed to 95 seconds for BWC (Figure 1).

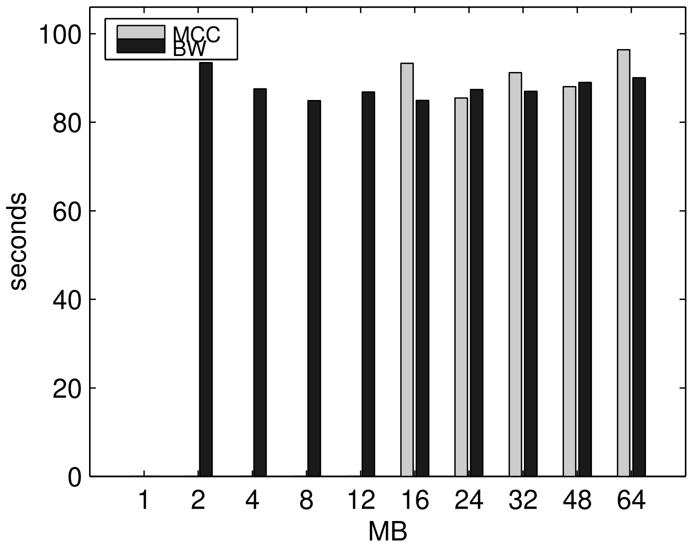

Fig. 2. Knuth-Bendix: Wall clock times as a function of heap size.

Knuth-Bendix This benchmark runs the Knuth-Bendix completion algorithm on a rewriting system. It allocates many 1, 2, and 3 word objects. Because of the many exception handlers it uses, the compiler is unable to take advantage of tail recursion, so Knuth-Bendix generates a very deep stack (over 4000 stack frames in some places).

The deep stack causes MCC to spend over half its time unpinning pages and scanning the stack. This is opposed to typical numbers of around 20% for other benchmarks. Since MCC uses a much slower algorithm to determine whether a value is a pointer than BWC, and as many as 5300 pointers from the stack are found per collection for this benchmark, BWC does better than MCC (Figure 2). Excessive pinning results in 1 MB being lost to fragmentation per collection. Because so many objects are pinned the `mark queue` is large for Knuth-Bendix. This in addition to the overhead for header words and copying explains why MCC needs so much more space than BWC for this benchmark [‖].

‖ A missing bar in the graph indicates that the associated collector could not run the benchmark in that space.

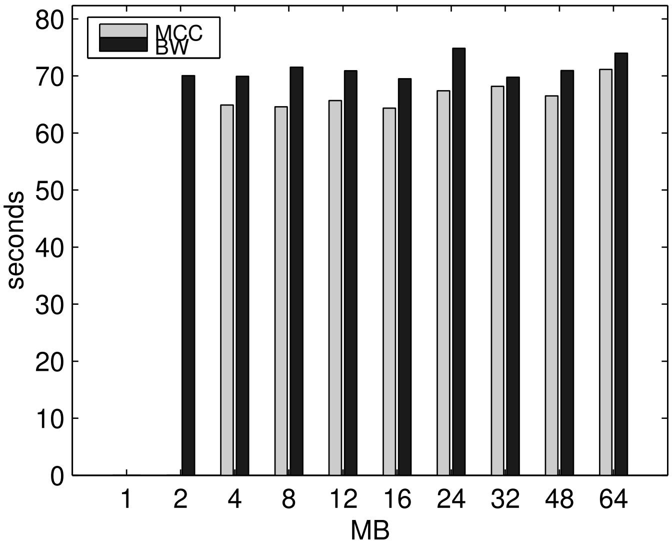

Fig. 3. Lexgen: Wall clock times as a function of heap size.

Lexgen This benchmark generates a lexer for ML. It is mostly I/O bound and allocates very little data, 20% of which is allocated atomically for BWC. The interesting fact to note about this benchmark is that MCC is not able to run it with one megabyte of memory whereas BWC is. This situation reflects the space needed to copy objects (Figure 3).

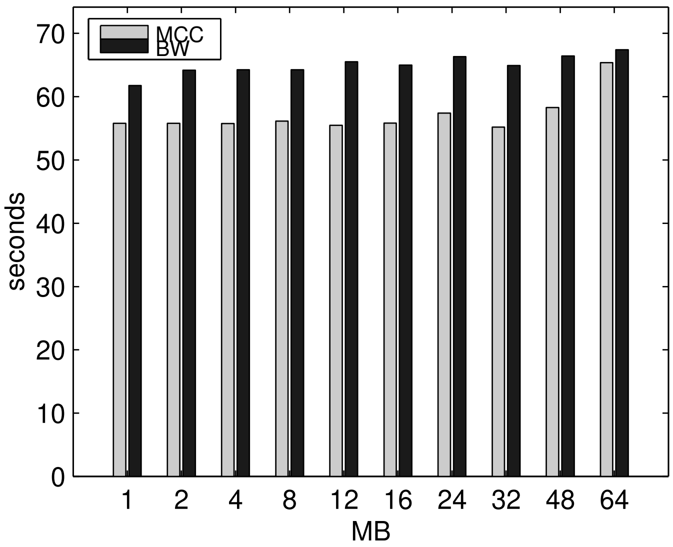

Fig. 4. Life: Wall clock times as a function of heap size.

Life This benchmark runs 10,000 generations of the Life simulation [23] for a small self-replicating automata. It allocates mostly lists, and pairs of integers. The pairs are allocated atomically for BWC and make up about 35% of the heap. On the other hand, there are typically only 4 KB of live data at any point in time. Just like Checksum, this enhances the advantages of copying collection. One interesting feature to note in the graph is that running times for MCC go up with increasing heap size and stay flat for BWC (Figure 4). We speculate that this is a combination of increased TLB misses for copying collection and possibly paging since both user and system time rise**.

** Complete data is available in Appendix A.

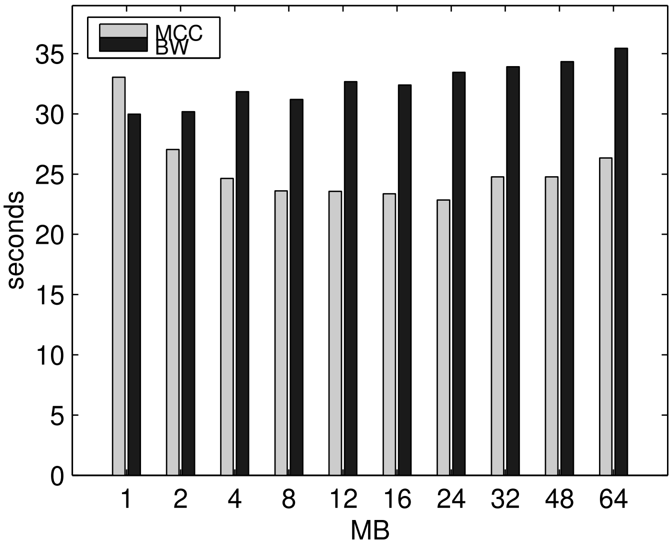

Fig. 5. Logic: Wall clock times as a function of heap size

Logic This benchmark is the output of a logic compiler, and performs some simple theorem proving using back-tracking and unification. The code allocates a mix of objects of different sizes. Logic has a shallow stack, and only about 30 KB of live data. These are conditions under which we expect MCC to do well, as the graph shows (Figure 5). MCC may also be benefiting from the mix of object sizes which forces BWC to maintain multiple lists.

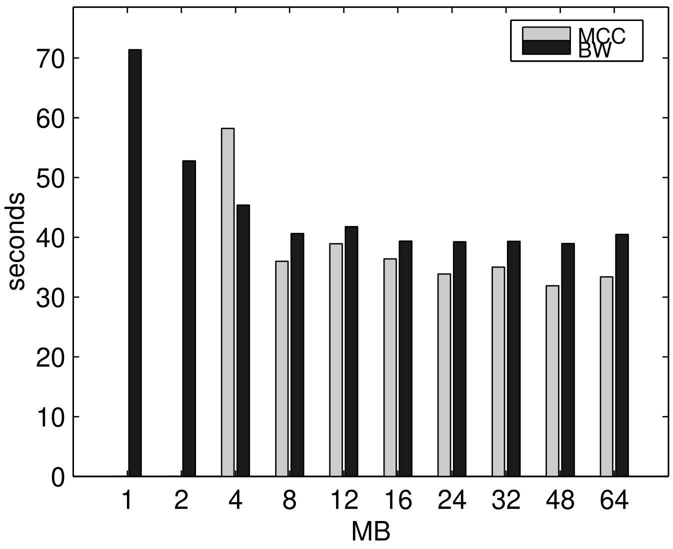

Fig. 6. Mergesort: Wall clock times as a function of heap size

Mergesort This benchmark sorts a list of 10,240 integers 100 times. The stack reaches depths of about 11, allocating new lists each time it splits the original list. This benchmark allocates 330 MB of cons-cells.

Mergesort does not have a deep stack, but at each collection there is a fairly large amount of live data. In addition, BWC gains the advantage of only maintaining information for elements of size two. MCC is paying the additional cost of a header word per object which amounts to a space overhead of 50%. This, in addition to the space needed for copying, explains why MCC can only run this benchmark with a 4 MB heap, whereas BWC runs it with only one. Lastly, MCC is doing about twice as many collections for a given heap size as BWC. These factors explain why MCC does not do as well on this benchmark as one might expect (Figure 6).

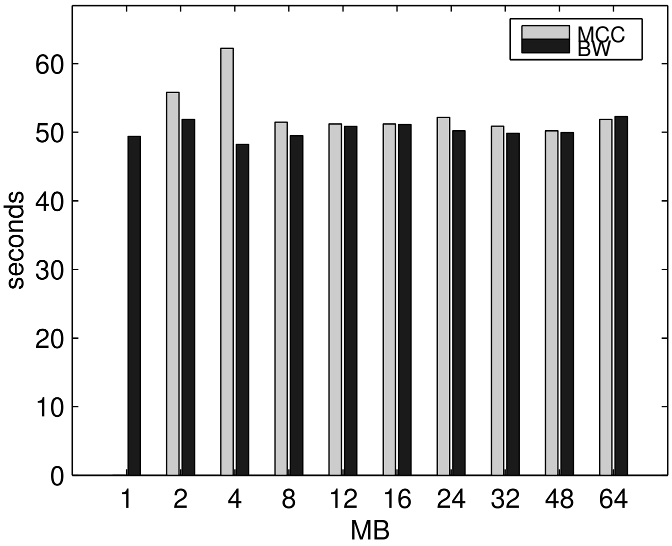

Fig. 7. Pia: Wall clock times as a function of heap size.

Pia This benchmark runs a perspective inversion algorithm that decides the location of an object in a perspective video image [31]. Most of the objects allocated are double-precision floating point values. The remainder (45%) are closures (3 words) and cons-cells (2 words). Since the floating point values are allocated atomically for BWC, the advantage of having type information is not as great for MCC. In addition, since floating point numbers have to be double word aligned this stresses code outside MCC's fast path that has not been optimized. This explains MCC's performance relative to BWC (Figure 7). There is a spike in the graph at 4 MB that we cannot explain, although it may be caused by an interaction with values on the stack.

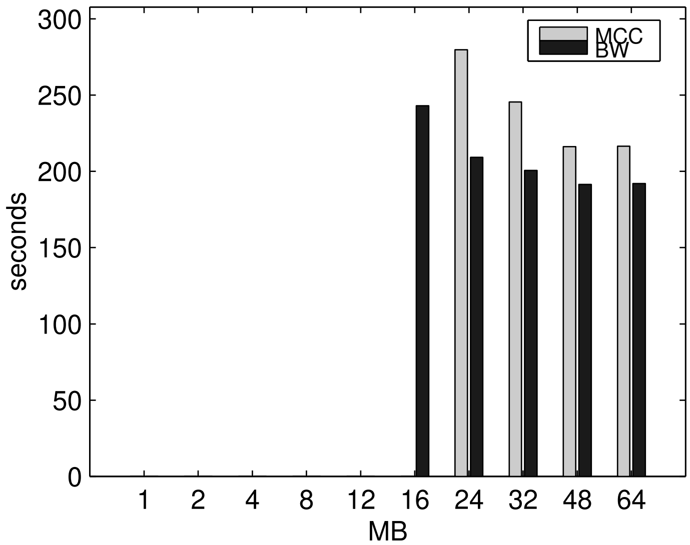

Fig. 8. Groebner: Wall clock times as a function of heap size.

Groebner This benchmark computes a Groebner basis [††]. It allocates only 2 word objects but has as much as 4 MB of live data (6 MB when we take header words into account). MCC aggressively sizes the heap by assuming that at most a third of the allocated data is live. Thus, MCC cannot run Groebner in 16 MB whereas BWC can. Lastly, Groebner uses polymorphism that the TIL compiler is unable to eliminate at compile time. Since we do not dynamically generate header words yet, Groebner is the only benchmark that has conservative objects in the heap (about 300 per collection). Because these objects also need to be double word aligned, this is the benchmark on which MCC does worst (Figure 8).

5 Future Work

There are several directions that the current work might take: enhancing the current collector with more sophisticated collection techniques, more detailed performance analyses with larger more realistic systems, detailed analyses of the various design decisions we made, and extending the basic ideas of mostly-copying collection to encompass other techniques. For example it would be interesting to modify the mostly copying collection algorithm to a mark-compact algorithm.

Currently we have a prototype implementation of generations for MCC that is not fully tuned. Initial results have not been promising but we intend to continue until we can explain them, or improve them.

[††] Groebner was provided to us by Thomas Yan.

We hope to do more detailed performance analyses in the future. We are particularly interested in how the language context affects collector performance, and are thus in the process of porting MCC to Toba [22], a Java compiler. We are also reworking TIL/C to support modules. Hence, we hope to have results for larger, longer-running programs in the near future.

On a more speculative note, mostly-copying collection may be well suited to a multi-threaded environment. In this setting BWC requires expensive locks to avoid contention for free lists, and even accurate collectors have trouble maintaining the required invariants. MCC could in principle hand out a different page to each thread, avoiding any need for synchronization and making allocation very fast. The primary challenge in this direction would be making the collector concurrent.

6 Conclusion

MCC demonstrates that a prototype mostly-copying collection can be competitive with a mature conservative mark-sweep collector. MCC does particularly well when allocation rates are high and the live data is a small fraction of the heap. However because it is a copying collector and because MCC uses header words it can require significantly more space than BWC. In addition, MCC implements a slow pointer testing routine which penalizes it further in the presence of a deep stack or many conservative objects. Despite these drawbacks, MCC is comparable if not better than BWC on almost all benchmarks. The study in [18] suggests that more sophisticated techniques such as generations, and segregating objects may yield another 20% improvement in collector times. And, a few modifications to our compiler may further decrease the cost of allocation. These factors lead us to speculate that in the presence of complete type information in the heap, MCC may someday be competitive with an accurate copying collector, not just conservative mark-sweep.

7 Acknowledgments

We extend our thanks to the many people who read and commented on this paper's earlier incarnations: Chris Hawblitzel, Stephanie Weirich, and David Walker.

References

1. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, Dec. 14 1989.
2. A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science.*
3. A. W. Appel and Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 1, January 1993.
4. G. Attardi and T. Flagella. Customising object allocation. In M. Tokoro and R. Pareschi, editors, *Object Oriented Programming, Proceedings of the 8th ECOOP, Lecture Notes in Computer Science 821*, pages 320–343. Berlin: Springer-Verlag, 1994.
5. G. Attardi and T. Flagella. A customizable memory management framework. In USENIX Association, editor, *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA*, pages 123–142, Berkeley, CA, USA, Apr. 1994. USENIX.
6. G. Attardi, T. Flagella, and P. Iglio. Performance tuning in a customizable collector. In H. Baker, editor, *Proceedings of International Workshop on Memory Management*, Lecture Notes in Computer Science, Dipartimento di Informatica, Universita di Pisa, Sept. 1995. Springer-Verlag.
7. J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, Feb. 1988.
8. J. F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical report, DEC WRL, Oct. 1989.
9. J. F. Bartlett. A generational, compacting collector for C++. In E. Jul and N.-C. Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1990.
10. J. F. Bartlett. System and method for garbage collection with ambiguous roots. US Patent, March 1990. Patent number 4907151. Assignee Digital Equipment Corporation.
11. H.-J. Boehm. Space-efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, June 1993.
12. H.-J. Boehm. Simple garbage-collector safety. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–98, Philadelphia, May 1996.
13. H.-J. Boehm and D. R. Chase. A proposal for garbage-collector-safe C compilation. *C Language Translation*, 1992.
14. H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.
15. J. Dean, G. DeFouw, D. Grove, V. Livinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. *ACM SIGPLAN Notices*, 31(10):83–100, Oct. 1996. Discusses performance of Vortex compiler for Cecil, C++, Java, and Modula-3.

16. A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, Jan. 1990.
17. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.
18. M. W. Hicks, J. T. Moore, and S. M. Nettles. The measured cost of copying garbage collection mechanisms. In *International Conference on Functional Programming*, Amsterdam, June 1997.
19. R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins. Reprinted February 1996.
20. G. Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995.
21. G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In *Usenix Conference on Object-Oriented Technlogies and Systems*, Oregon, June 1996.
22. T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. Technical Report TR97-01, The Department of Computer Science, University of Arizona, Jan. 8 1997. Wed, 08 Jan 97 00:00:00 GMT.
23. C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, 1989.
24. M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. *Lecture Notes in Computer Science*, 983:366–81, 1995.
25. Z. Shao and A. W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, pages 150–161, Orlando, June 1994.
26. Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.
27. B. Stroustrup. *The C++ programming language.* Addison-Wesley series in computer science. Addison-Wesley, Reading, MA, USA, reprinted with corrections edition, 1987.
28. D. Tarditi. *Design and Implementation of Code Optimizations for a TypeDirected Compiler for Standard ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1996.
29. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, Pennsylvania, 21– May 1996.
30. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 31(5):181–192, May 1996.
31. K. G. Waugh, P. McAndrew, and G. Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, Aug. 1990.
32. P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, Sept. 1992. Springer-Verlag.

33. B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.

A Data & Methodology

We ran all the experiments on a loaded Sparc 20 running Solaris 2.5 with 256 MB of memory.

The statistics in Table 2 were collected by running MCC for the smallest heap size that compiled that particular benchmark. Then the percentage of the time spent in each phase was computed for each collection and the result averaged over all the collections. The results do not add up to one hundred percent because the timer overhead is so great for some of the faster running collections.

The data in Table 3 were collected by averaging the appropriate values over the same runs as above. Basically, they record the average number of roots, live objects, and words lost to fragmentation per collection.

The remaining tables contain the timing data. We chose to plot wall clock times because these were fairly representative, but all the timing data is present here. The results were generated by averaging ten runs of each program where the worst run has been omitted for each case. For a given heap size and benchmark, BWC and MCC were run in succession so they should exhibit similar patterns reflecting the load on the machine when they were run. When no data is available for a particular heap size, it is because that collector was not able to run the benchmark within the required memory.

The difference between overall time and collector time should be interpreted differently for BWC and MCC. For BWC we did not time lazy-sweeping because the overhead of the timer was too great. This means that for BWC (overall time−collector time) includes the time for lazy-sweeping in addition to the time for the client and allocation. In contrast, for MCC, (overall time − collector time) includes only client and allocation time.

Benchmark	Pin Children	Root Scan	Graph Trav.	Free Pages	Unpin Pages	Unaccounted For
Checksum	2.89	17.41	4.09	6.48	5.58	63.55
Knuth-Bendix	0.06	44.93	37.04	1.21	15.18	1.58
Lexgen	0.29	5.67	83.54	1.25	2.43	6.82
Life	2.10	22.97	10.12	5.50	7.76	51.55
Logic	0.83	44.85	16.52	2.74	16.42	18.64
Mergesort	0.21	3.07	88.42	1.40	1.88	5.02
Pia	0.82	22.28	44.67	3.87	8.34	20.02
Groebner	17.49	0.78	77.34	0.77	2.84	0.78

Table 2. Breakdown of typical collection times.

Benchmark	Average number of		
	Roots	Live Objects	Wasted Words
Checksum	35	46	1092
Knuth-Bendix	5301	90280	259875
Lexgen	106	32364	16027
Life	57	556	2020
Logic	264	2683	11654
Mergesort	61	63523	3482
Pia	319	7065	9667
Groebner	131	338069	12214

Table 3. Statistics gathered during collection.

Heap Size(MB)		Total Time			GC Time		
		User	Sys	Wall	User	Sys	Wall
1	BWC	90.75	1.06	96.90	10.48	0.32	11.23
	MCC	60.93	1.59	65.74	4.25	0.54	4.89
2	BWC	97.10	0.80	101.36	7.22	0.19	7.69
	MCC	64.12	0.92	67.56	2.43	0.27	2.74
4	BWC	96.19	0.80	104.68	5.48	0.11	6.07
	MCC	63.54	0.73	68.93	1.39	0.13	1.61
8	BWC	97.49	1.07	104.01	4.52	0.07	4.80
	MCC	62.45	0.79	66.76	0.87	0.07	0.99
12	BWC	95.24	1.41	105.47	4.22	0.05	4.61
	MCC	62.74	0.98	69.89	0.72	0.05	0.80
16	BWC	96.88	1.77	106.00	4.08	0.04	4.37
	MCC	62.42	1.21	67.20	0.63	0.03	0.69
24	BWC	95.50	2.57	103.72	4.09	0.03	4.33
	MCC	62.90	1.73	69.11	0.55	0.03	0.65
32	BWC	95.18	3.32	105.33	4.05	0.03	4.47
	MCC	62.82	2.22	69.11	0.51	0.02	0.57
48	BWC	94.23	4.78	103.05	4.13	0.02	4.31
	MCC	61.96	3.32	68.03	0.46	0.01	0.48
64	BWC	93.97	6.31	102.86	4.10	0.02	4.24
	MCC	61.55	4.56	68.45	0.45	0.01	0.45

Table 4. Checksum: Timing data

Heap Size(MB)		Total Time			GC Time		
		User	Sys	Wall	User	Sys	Wall
2	BWC	87.40	0.51	93.49	10.84	0.08	11.59
4	BWC	81.57	0.70	87.53	5.09	0.05	5.49
8	BWC	78.45	1.09	84.88	2.30	0.04	2.45
12	BWC	78.08	1.49	86.84	1.79	0.03	1.89
16	BWC	77.19	1.90	84.94	1.21	0.02	1.40
	MCC	85.69	1.60	93.33	9.23	0.26	10.30
24	BWC	80.25	2.79	87.41	1.37	0.02	1.44
	MCC	78.86	1.99	85.48	3.12	0.06	3.46
32	BWC	77.20	3.60	86.98	0.78	0.02	0.84
	MCC	80.12	2.73	91.20	3.81	0.13	4.28
48	BWC	77.63	5.39	89.03	0.90	0.01	0.98
	MCC	79.50	3.95	88.07	1.99	0.08	2.14
64	BWC	78.41	7.17	90.06	0.56	0.01	0.65
	MCC	85.78	5.50	96.38	2.41	0.15	2.70

Table 5. Knuth-Bendix: Timing data

Heap Size(MB)		Total Time			GC Time		
		User	Sys	Wall	User	Sys	Wall
2	BWC	65.85	0.91	70.07	3.27	0.03	3.37
4	BWC	64.39	1.11	69.95	1.29	0.02	1.35
	MCC	60.54	0.98	64.89	3.74	0.09	4.06
8	BWC	64.13	1.55	71.52	0.89	0.01	1.01
	MCC	58.84	1.28	64.59	1.75	0.08	1.94
12	BWC	63.17	1.94	70.89	0.59	0.01	0.65
	MCC	58.06	1.55	65.69	1.14	0.09	1.39
16	BWC	62.88	2.36	69.49	0.56	0.01	0.61
	MCC	57.96	1.81	64.35	0.82	0.07	0.96
24	BWC	63.29	3.31	74.85	0.43	0.01	0.46
	MCC	57.63	2.46	67.40	0.50	0.06	0.58
32	BWC	62.21	4.14	69.79	0.29	0.01	0.30
	MCC	58.13	3.13	68.18	0.35	0.07	0.42
48	BWC	61.54	5.79	70.96	0.21	0.00	0.22
	MCC	57.72	4.27	66.52	0.12	0.04	0.16
64	BWC	62.14	7.61	73.99	0.30	0.00	0.31
	MCC	59.49	5.73	71.15	0.26	0.06	0.41

Table 6. Lexgen: Timing data

Heap Size(MB)		Total Time			GC Time		
		User	Sys	Wall	User	Sys	Wall
1	BWC	58.80	0.38	61.77	2.64	0.08	2.82
	MCC	51.89	0.54	55.80	2.37	0.16	2.68
2	BWC	60.35	0.41	64.18	1.79	0.05	1.93
	MCC	52.78	0.42	55.77	1.21	0.08	1.32
4	BWC	60.15	0.57	64.26	1.35	0.03	1.45
	MCC	52.07	0.45	55.74	0.66	0.04	0.72
8	BWC	60.09	0.93	64.25	1.11	0.02	1.26
	MCC	51.78	0.67	56.14	0.37	0.02	0.39
12	BWC	59.86	1.31	65.52	1.00	0.01	1.10
	MCC	51.42	0.89	55.45	0.28	0.01	0.30
16	BWC	59.72	1.67	64.99	1.03	0.01	1.08
	MCC	51.73	1.17	55.83	0.24	0.01	0.25
24	BWC	60.02	2.52	66.33	1.01	0.01	1.13
	MCC	51.64	1.74	57.40	0.19	0.01	0.21
32	BWC	59.79	3.25	64.92	0.91	0.01	0.98
	MCC	51.43	2.25	55.18	0.17	0.01	0.19
48	BWC	59.36	4.82	66.43	0.98	0.01	1.05
	MCC	51.80	3.45	58.27	0.15	0.00	0.17
64	BWC	59.23	6.39	67.41	0.79	0.01	0.79
	MCC	58.55	4.70	65.38	0.14	0.00	0.14

Table 7. Life: Timing data

Heap Size(MB)		Total Time			GC Time		
		User	Sys	Wall	User	Sys	Wall
1	BWC	28.71	0.31	29.98	2.44	0.06	2.55
	MCC	30.82	0.49	33.04	11.44	0.17	12.28
2	BWC	28.77	0.35	30.18	1.57	0.04	1.62
	MCC	25.44	0.34	27.04	5.18	0.07	5.56
4	BWC	29.14	0.53	31.83	1.14	0.02	1.21
	MCC	22.98	0.40	24.64	2.56	0.04	2.69
8	BWC	28.75	0.94	31.20	0.89	0.02	0.95
	MCC	21.70	0.63	23.62	1.26	0.02	1.34
12	BWC	28.71	1.31	32.68	0.79	0.01	0.87
	MCC	21.22	0.87	23.58	0.85	0.01	0.91
16	BWC	28.53	1.68	32.39	0.72	0.01	0.78
	MCC	20.82	1.14	23.38	0.65	0.01	0.67
24	BWC	28.81	2.53	33.45	0.73	0.01	0.77
	MCC	20.53	1.71	22.86	0.43	0.01	0.44
32	BWC	28.72	3.34	33.91	0.67	0.01	0.71
	MCC	20.49	2.33	24.76	0.32	0.01	0.35
48	BWC	28.34	4.93	34.32	0.61	0.01	0.63
	MCC	20.35	3.45	24.77	0.23	0.01	0.24
64	BWC	28.01	6.52	35.44	0.53	0.01	0.60
	MCC	20.87	4.75	26.34	0.19	0.00	0.21

Table 8. Logic: Timing data

Heap Size(MB)		Total Time User	Sys	Wall	GC Time User	Sys	Wall
1	BWC	68.08	0.64	71.38	37.56	0.24	38.97
2	BWC	50.53	0.49	52.78	18.34	0.11	19.05
4	BWC	42.04	0.57	45.39	9.41	0.06	10.00
	MCC	54.74	0.66	58.24	26.66	0.24	28.29
8	BWC	37.62	0.90	40.64	5.06	0.03	5.49
	MCC	32.79	0.72	35.99	6.77	0.11	7.54
12	BWC	36.33	1.27	41.79	3.77	0.02	4.23
	MCC	33.59	0.96	38.92	6.90	0.11	8.06
16	BWC	35.27	1.66	39.37	2.82	0.02	3.07
	MCC	33.03	1.22	36.38	6.47	0.12	6.98
24	BWC	34.91	2.45	39.25	2.37	0.01	2.48
	MCC	30.38	1.72	33.88	3.97	0.10	4.30
32	BWC	34.46	3.21	39.34	2.20	0.01	2.31
	MCC	29.29	2.24	35.01	2.97	0.09	3.34
48	BWC	33.47	4.71	38.96	1.69	0.01	1.77
	MCC	27.92	3.36	31.90	2.00	0.09	2.14
64	BWC	33.37	6.20	40.50	1.73	0.01	1.81
	MCC	27.89	4.55	33.40	1.50	0.09	1.63

Table 9. Mergesort: Timing data

Heap Size(MB)		Total Time User	Sys	Wall	GC Time User	Sys	Wall
1	BWC	45.22	0.80	49.39	3.45	0.08	3.67
2	BWC	46.30	0.86	51.86	2.63	0.05	2.84
	MCC	51.04	0.86	55.83	6.35	0.11	6.82
4	BWC	44.25	1.00	48.21	1.52	0.03	1.62
	MCC	56.53	0.93	62.23	11.73	0.09	12.70
8	BWC	44.00	1.37	49.48	1.16	0.02	1.26
	MCC	46.79	1.06	51.46	2.33	0.04	2.51
12	BWC	43.91	1.75	50.86	1.04	0.02	1.15
	MCC	45.36	1.33	51.23	1.23	0.03	1.37
16	BWC	43.78	2.19	51.12	1.00	0.01	1.15
	MCC	45.48	1.56	51.20	0.91	0.02	1.01
24	BWC	43.93	2.98	50.20	0.84	0.01	0.89
	MCC	46.19	2.10	52.15	1.02	0.02	1.08
32	BWC	43.66	3.74	49.84	0.83	0.01	0.85
	MCC	45.16	2.70	50.89	0.62	0.02	0.68
48	BWC	43.02	5.33	49.94	0.80	0.01	0.81
	MCC	44.65	3.85	50.19	0.48	0.02	0.51
64	BWC	43.28	6.92	52.29	0.81	0.01	0.82
	MCC	44.72	5.11	51.85	0.24	0.01	0.27

Table 10. Pia: Timing data

Heap Size(MB)		Total Time User	Sys	Wall	GC Time User	Sys	Wall
16	BWC	213.41	2.32	243.03	43.29	0.12	47.91
24	BWC	195.27	3.02	209.20	29.03	0.07	30.63
	MCC	260.06	2.98	279.69	105.87	0.91	113.10
32	BWC	189.91	3.76	200.57	22.21	0.05	23.22
	MCC	235.57	3.20	245.47	80.88	0.62	83.72
48	BWC	181.80	5.30	191.34	16.29	0.04	16.81
	MCC	206.37	4.36	216.23	54.42	0.59	56.61
64	BWC	179.04	6.99	192.00	12.59	0.03	12.99
	MCC	202.76	5.79	216.42	42.65	0.60	44.95

Table 11. Groebner: Timing data