Comparing Mostly-Copying and Mark-Sweep Conservative Collection *

Frederick Smith †
Cornell University
fms@cs.cornell.edu

Greg Morrisett
Cornell University
jgm@cs.cornell.edu

Abstract

Many high-level language compilers generate C code and then invoke a C compiler for code generation. 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 an extension of *mostly-copying collection*.

We analyze the various design decisions made in MCC and provide a performance comparison to the most widely used conservative mark-sweep collector (the Boehm-Demers-Weiser collector). Our results show that a good mostly-copying collector can outperform a mature highly-optimized mark-sweep collector when physical memory is large relative to the live data. A surprising result of our analysis is that cache behavior can have a greater impact on overall performance than either collector time or allocation time.

1 Overview

For almost any language, there are a handful of compilers that generate C [1] as a target language. For instance, Java-to-C compilers include Toba [20], Vortex [11], and Harissa [18]; and ML-to-C compilers include Bigloo [22], SML2C [27], and TIL/C [28][1]. The developers of these compilers chose C as a target language to leverage the optimizations of existing C compilers and to obtain a relatively portable, easy-to-build back-end.

However, Java, ML, and many other languages require garbage collection which C does not directly provide. Because of the difficulties of combining C with *accurate* garbage collection, all of the systems mentioned above (except SML2C) are designed to work with a *conservative* garbage collector, and in fact, use the Boehm-Demers-Weiser, mark-sweep conservative collector (BDW). BDW is conservative because it "guesses" which values are pointers without any annotations. In other respects BDW is similar to a highly tuned, accurate mark-sweep collector: it potentially suffers the same fragmentation problems, and relative to copying collectors, requires more instructions for allocation.

An alternative to conservative mark-sweep, first proposed by Bartlett [5] and ideally suited to compilers from type safe languages to C, is mostly-copying collection. Mostly-copying collection requires that most objects be typed (annotated to distinguish pointers from other values) in order to be effective. Under this assumption, mostly-copying collection can share the same characteristics as accurate copying collection: fast allocation and compaction. Mostly-copying also suffers from the main disadvantage of copying collection – greater space requirements.

In our work with the TIL/C SML-to-C compiler, mostly-copying collection seemed promising because we have type information for almost all objects in the heap, and it is widely thought that copying collection is the right strategy for ML[2]. In particular, ML programs tend to allocate a lot of short-lived data and so fast allocation seemed relatively important.

Our collector, MCC, extends Bartlett's original algorithm by allowing typed and untyped objects to coexist in the heap, and by making only one traversal of the live data. MCC aims to make allocation fast and achieve good collector performance, without sacrificing the benefits of compiling to C.

To evaluate MCC, we measured the performance of eight small to medium sized ML benchmarks run with both BDW and MCC. To isolate as many variables as possible, we used a non-generational version of MCC with a simple collection policy – collect whenever the heap becomes two-thirds full. The data show that when given enough space, MCC collection times are significantly faster than for BDW. Surprisingly, MCC gives only slightly better overall (client + collector) performance. We assumed that client times for MCC would *always* be faster than for BDW, because the MCC allocation sequence requires fewer instructions, and fewer load/store instructions. To our surprise, we found that in many cases, the client was actually slowed under MCC due to a loss of spatial locality. Contrary to our initial belief, a fast allocation sequence was relatively unimportant for overall performance.

The main results of this paper are:

*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.

[1] This list is not exhaustive.

[2] Both SML/NJ [2] and O'Caml [19], two widely-used ML compilers, use variants of copying collection for at least the most recently allocated data.

- A new one-pass mostly-copying collector, MCC, that allows typed and untyped objects to co-exist in the heap and to be collected. Previous algorithms have either been two-pass or have treated untyped objects as residing outside the heap.
- An analysis of the design decisions made in MCC and their impact on overall collector performance.
- A comparison between BDW and MCC in the context of an ML-to-C compiler.

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 Paul Wilson's survey on garbage collection techniques [31] or Richard Jones' text [16].

2 Conservative Collection

Conservative garbage collection is the extension of accurate garbage collection to environments where partial or no type information is available. In particular, a conservative collector is not told which values are pointers for some subset of the heap. We refer to objects without pointer information as *untyped* objects.

In our setting untyped objects arise from the stack (because the C compiler does the stack layout) and polymorphism (because we do not yet dynamically generate type information). However in general, untyped objects may arise from other sources. Some examples include C code that has been linked against, dynamic libraries, and continuations.

To discover the live graph, the collector must conservatively determine which values in untyped objects represent pointers. To make this feasible, conservative collectors require that programs maintain at least one pointer to every live object. In particular, programs that hide or manufacture pointers are not supported. We call a value that is inferred, but not known, to be a pointer a *quasi-pointer* (sometimes referred to as an ambiguous pointer [6]).

Misidentification of values as quasi-pointers artificially increases the size of the live graph, and limits the effectiveness of the collector. Of course correctly distinguishing pointers from other values is not always possible, and for some large, long running programs pointer misidentification may be a significant problem [14]. However, for most programs it is possible to detect quasi-pointers with great accuracy (usually less than 10% misidentification) and speed (30 instructions) [16]. This is possible because of architectural constraints that force pointers to be multiples of the word-size and to reside in specific address ranges [10, 9].

Conservative collectors have traditionally been limited to collection algorithms that do not update values in the client's data. Since quasi-pointers are not known to be pointers, altering a quasi-pointer to point to a new location, as would be necessary in copying collection, could change the client's behavior. For this reason, most work on conservative collection has focused on mark-sweep algorithms.

However in certain settings much type information is available. This is typically the case when safe language compilers produce C code. The compiler has exact information for all objects in the heap, but does not control the stack layout. In these situations a variant of copying collection called mostly-copying collection is possible.

In the following subsections, we examine the algorithms and implementations of the BDW mark-sweep collector and the MCC mostly-copying collector in greater detail.

2.1 The Boehm-Demers-Weiser Collector

BDW [10, 8] is a conservative mark-sweep collector available on a wide variety of platforms. It is used by at least 13 language implementations to date and has been under development since 1987[3]. BDW 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] (for programs with small heaps BDW may require significantly more memory). For better performance, clients may allocate pointer-free (*atomic*) data through a special allocation routine – `GC_malloc_atomic`.

Except for quasi-pointer detection, the BDW collection algorithm is a highly-optimized version of mark-sweep. Some of its more interesting, and relevant features are:

1. Objects are segregated based on size and whether they are atomic or not. Separate free lists are maintained for each kind of object. Each page of the heap holds objects of a single size. This speeds up quasi-pointer detection by making it easy to find the beginning of an object. Large objects are handled differently.
2. Inline allocation. Macros for inline allocation are provided. On a Sparc the allocation code for the fast path takes 9 instructions.
3. Lazy-sweeping. Instead of sweeping the whole space after each collection, the collector incrementally sweeps the space whenever the free list becomes exhausted until the sweep is complete. Only then is a collection invoked to refill the free list.

BDW can detect quasi-pointers very effectively. Most failed pointer tests take about 5 instructions, and a successful quasi-pointer detection takes about 30 instructions [16]. The speed of pointer testing makes completely conservative collectors such as BDW practical.

2.2 MCC

Mostly-copying collection is an idea originally due to Bartlett [5]. As discussed above, a sound collector cannot modify a quasi-pointer and cannot therefore move an object referenced by a quasi-pointer. In fact exactly those objects which are not referenced by quasi-pointers can be copied. Consequently, if the untyped portion of the heap is small, and there are few quasi-pointers, a collector can copy most objects.

To implement this idea efficiently requires one key modification to the copying collection algorithm. Instead of using contiguous blocks of memory for to-space and from-space, the algorithm uses a linked list of *pages*. To "copy" an object that cannot be moved, the collector links the page on which the object resides into the to-space list.

MCC is a particular implementation of mostly-copying collection. We break the discussion of MCC into three parts. The first part presents an abstract view of the algorithm. The second part discusses some practical implementation issues and design decisions. The third part compares MCC's algorithm to related work.

2.2.1 The Algorithm

MCC is a hybrid mark-sweep, copying collector that supports both typed and untyped objects. All objects include

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

meta-data for collection purposes. The meta-data include the size of the object, and two state bits used during garbage collection. One state bit determines whether the object is treated via mark-sweep or copying collection. The other state bit determines whether the object has been visited. For typed objects, the meta-data also include information indicating which values in the object are pointers.

The collector breaks memory into fixed-size *pages* which are organized by free lists. When the collector determines that the number of free pages is below some threshold, it initiates garbage collection.

The algorithm begins by scanning all sources of quasi-pointers (including the roots and any untyped objects in the heap) and *pins* all objects referenced by these quasi-pointers (by toggling one state bit in each object). These are exactly the objects that cannot be moved and that will be treated via mark-sweep collection. Note that untyped objects may be copied if they are not referenced by any quasi-pointers.

In the second stage, a hybrid mark-sweep/copying collection begins. The collector maintains two sets of pages: a set of from-space pages and a set of to-space pages. As in a copying collector, most live objects are copied from pages in from-space to pages in to-space. However, live pinned objects cannot be copied. Hence, the page on which they reside is *promoted* to to-space.

The collector maintains both a mark-queue and a Cheney-queue [32]. When a pinned object is first encountered, it is placed in the mark-queue, whereas other objects are forwarded and placed in the Cheney-queue. The algorithm begins by scanning the roots and placing the objects they reference in the appropriate queue. Objects are taken off either queue and processed until both queues are empty.

When the queues become empty, all reachable objects have been visited, and all pages remaining in from-space may be reclaimed. The collector then clears the state bits and sweeps the set of unmarked objects on promoted pages.

The sweep phase is important. In a conservative setting it is *unsound* to leave dead objects with dangling pointers on promoted pages because a dead object may later be found (incorrectly) by a quasi-pointer. If the object is typed, then the collector assumes that all of its pointers reference preserved objects, which may not be true. Therefore a mechanism is needed to explicitly mark the swept space as invalid.

We separate the algorithm into the following stages:

- *Pin.* Pin all the objects referred to by quasi-pointers.
- *Root.* Scan the roots, and begin graph traversal.
- *Graph.* Perform the graph traversal described above.
- *Free.* Reclaim the pages left in from-space.
- *Sweep.* Sweep all the promoted pages, and clear the state bits for the marked objects.

The phases pin and sweep each touch all pinned objects once. Graph traversal touches both copied and mark-swept objects twice – once on the first visit, and the second time to process the object's children. Therefore this algorithm runs in time proportional to $O(2n + 2m + p)$ where n is the size of the live data, m is the number of pinned objects, and p is the number of pages reclaimed. In our setting, m tends to be very small as is the constant associated with p, and hence collection is typically proportional to the size of the live data.

2.2.2 Design Details

Our implementation of the above algorithm is designed with one overarching aim: fast allocation. Several studies in the literature have suggested that allocation costs can be significant [13, 12, 15], and are in fact one of the main advantages of copying collection. We were interested in determining whether we could reap these same benefits in a mostly-copying setting.

To achieve fast allocation, clients allocate objects contiguously on a single page until the page is full. Thus MCC achieves the same amortized cost for allocation as in a copying collector. Since all types of objects are allocated together, this decision precludes segregating objects based on size, and requires the use of a header word on every object to identify its size, type, and state. Because we wanted to allow untyped objects within the heap we also chose not to tag pointers.

Mixing objects of various sizes together complicates quasi-pointer detection. Once a value is found to point to a particular page, the collector needs to determine what specific object is being referenced. If the objects are segregated based on size, simple arithmetic suffices. Otherwise, the collector must scan from the beginning of the page until the appropriate object is found. Consequently, MCC successfully detects a quasi-pointer in the average case with time proportional to half the number of objects on a page (about 100 instructions for 2 KB pages of cons-cells).

MCC allows untyped objects to be scattered throughout the heap. Therefore, the collector maintains a list of all untyped objects in the heap so that it can find all quasi-pointers in the pin stage of the algorithm. An alternative solution would have been to segregate untyped objects.

Since whenever an object is marked the collector must promote the whole page, MCC pins all objects when it promotes a page. This avoids unnecessarily copying objects that are already logically in to-space. After sweeping, promoted pages may have unused space. MCC does not reuse this space in subsequent collections.

MCC correctly deals with large objects, alignment restrictions, and pointers from and into the static region. Readers interested in further details should refer to our technical report [25].

2.2.3 Related Collectors

There are two other mostly-copying collectors that bear a strong resemblance to MCC. One, due to Bartlett, is used in a Scheme-to-C compiler and in its newest variant has migrated to C++ [7]. The other is the Customizable Memory Manager (CMM) which is also a collector for C++ [3, 4].

Bartlett's algorithm does two traversals over the live graph. Conceptually, in the first pass the algorithm identifies all quasi-pointers and all objects that cannot be moved. It promotes all pages containing such objects to to-space by linking them into the to-space list. In the second pass, the collector performs a standard copying collection using the objects promoted into to-space as roots. Because all roots are untyped, all objects referred to by roots get promoted. Therefore, it is correct to only use promoted objects as roots during the second pass. As an optimization, introduced by CMM, the collector remembers the live objects from the first pass so that it does not retain dead objects that happen to reside on a promoted page.

The algorithm that Bartlett actually uses aggressively copies all objects during the first pass and unrolls copies that

Checksum	Performs a checksum on a stream of 16-bit values.
Knuth-Bendix	Runs the Knuth-Bendix completion algorithm on a rewriting system.
Lexgen	Generates a lexer for ML.
Life	Runs $10,000$ generations of the Life simulation [21] for a small self-replicating automata.
Logic	Performs some simple theorem-proving using back-tracking and unification.
Mergesort	Mergesorts a large list repeatedly.
Pia	Runs a perspective inversion algorithm to decide the location of an object in a perspective video image [30].
Groebner	Computes a Groebner basis.

Table 1: Brief description of the benchmarks.

	Heap (MB)	Alloc. (MB)	Live (KB)	# of GCs	GC (ms)	Live Obj./GC
Checksum	8	979	4	262	1	46
KB	16	103	1136	15	161	84.732
Lexgen	8	75	503	19	28	28,374
Life	8	244	14	69	1	501
Logic	8	173	30	47	5	2,951
Mergesort	8	328	262	98	13	26,919
Pia	8	213	111	68	8	6,637
Groebner	32	1018	3713	92	210	335,480

Table 2: Basic information: the canonical heap size (CHS) used for other measurements, total allocation, most live data after any collection, # of collections (at CHS), how long each collection took (average), live objects per collection (average).

were incorrectly made (because the object was referenced by a quasi-pointer) during the second pass.

CMM is a customizable collector that supports mostly-copying collector. BDW and CMM have been compared in the context of C++ [4]. Like Bartlett, CMM implements a two stage mostly-copying algorithm. In the first stage all other heaps are examined for quasi-pointers, or pointers into this heap. All pages containing objects referenced by quasi-pointers are promoted into to-space. The heap being collected does not contain any untyped objects.

In the second stage, a copying collection begins using as roots the stack, static area, and all the objects found during the first stage. One of the innovations introduced by CMM was the *live map* – a bitmask associated with each page that records the objects that are actually live on that page. Before this development, collectors treated all objects on promoted pages as roots with false retention in excess of 50% [3].

This algorithm is proportional to the size of the other heaps, and the live data in the collected heap. Untyped objects exist in a different heap and are collected via some other collection strategy. One of the drawbacks of CMM's approach is that it cannot break cycles between heaps.

3 Empirical Results

3.1 Comparison of Overall Costs for ML

To compare MCC to BDW, we added a C back end to the TIL compiler [29, 28] and targeted it to use both collectors. TIL generates fairly natural C code that uses the C stack and produces natural C constructs such as while loops.

We compiled eight small to medium sized benchmarks for ML, using BDW 4.10 and then MCC. The code, except for allocation, collection, and the array size primitive, is the same for both versions. We selected benchmarks that have been used in the literature [17, 26, 24, 23] to compare the performance of TIL and SML/NJ [2]. Table 1 gives a brief description of each benchmark[4]. 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 UltraSparc-5 running Solaris 2.6[5].

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

[5]A complete explanation of our methodology is given in Appendix A

To improve BDW's performance we inlined almost all allocations, allocated pointer free objects atomically, configured BDW for large heaps, and disabled interior pointers.

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. There was only one benchmark (Groebner) with untyped objects in the heap.

We instrumented both collectors with high resolution timers to time overall running times and collection times. Collection times do not include allocation but do include all other collection work including lazy-sweeping for BDW. We collected timing information for a variety of heap sizes ranging from 1 MB up to 64 MB. Both collectors were initialized to start at the measured size, and constrained to keep their memory consumption within this size.

Figure 1 shows the empirical results[6]. At the end of this section we analyze each benchmark individually to explain its performance. In the following subsection we summarize those analyses along three lines: client time, collection time, and space requirements. Some basic information about each benchmark is shown in Table 2.

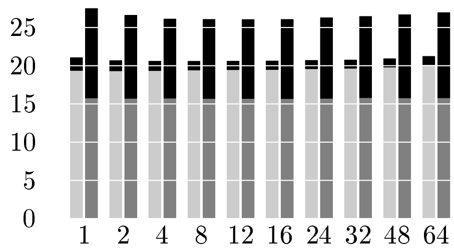
3.1.1 Client Time

The client times vary by as much as 20% between the two collectors. Although the client code for MCC typically requires fewer instructions because of MCC's shorter allocation sequence, MCC's client times are in many cases worse than BDW's. There are four effects that we believe account for the variation in client times:

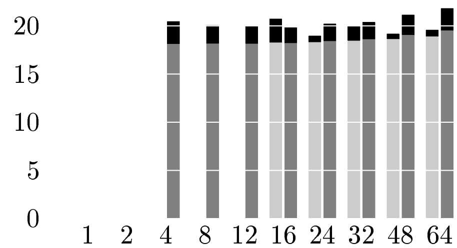
- *Headers.* For programs that predominantly allocate data of a single size, BDW and MCC will for practical purposes layout the data sequentially. If in addition, the access pattern is similar to the allocation pattern, MCC has worse cache performance because of the additional header word. A common example for ML that fits this pattern are list elements of two words. Instead of accessing the data with stride two, a benchmark using MCC is accessing it with stride three.

  To confirm that this was the effect we observed we added a word to each object allocated by BDW and

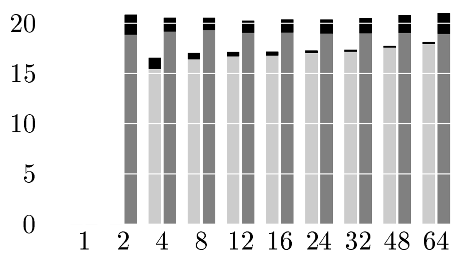
[6]The actual numbers for all the experiments are available in Appendix A.



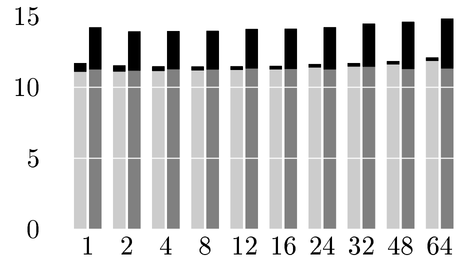
(a) Checksum



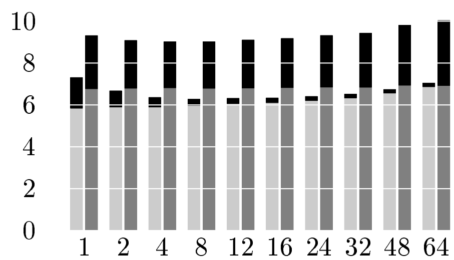
(b) Knuth-Bendix



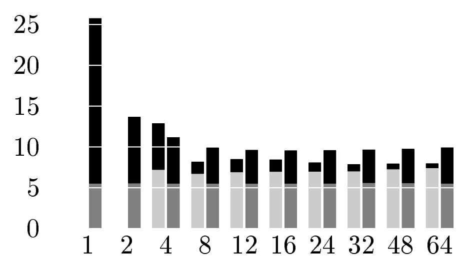
(c) Lexgen



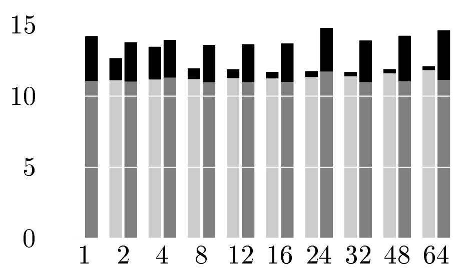
(d) Life



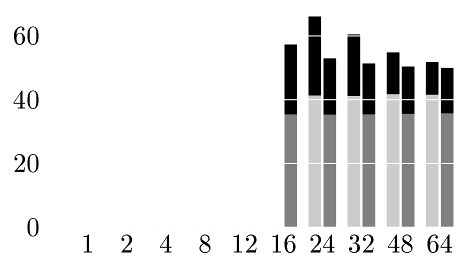
(e) Logic



(f) Mergesort



(g) Pia



(h) Groebner

Figure 1: The vertical axis shows wall clock times (seconds) averaged over ten runs as a function of heap size (MB, horizontal axis). MCC (light) and BDW (dark) client times are shown, in addition to the respective collector times (black). A missing bar indicates that the benchmark could not be run at that heap size.

Figure 2: Percentage slowdown relative to BDW client times for MCC client times (light) and BDW with an extra word on each object (dark).

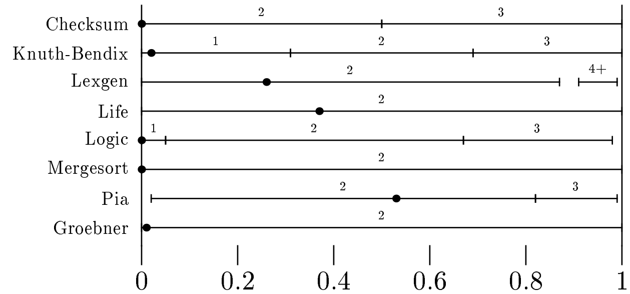


Figure 3: Percentage of objects allocated of size 1, 2, 3, and 4+. The dot indicates the percentage of the data that was pointer-free. (Values less than 5% have been omitted.)

measured the client time slowdown. Figure 2 compares the observed slowdown for MCC to the slowdown for BDW with the extra word. As expected, for Mergesort and Groebner, which allocate only list pairs (see Figure 3), the measured slowdown is similar. Figure 2 is not a good predictor for the other benchmarks because multiple object sizes (or atomic data) are involved.

- *Size.* BDW maintains two free lists for each size of object – one for pointer-free objects and one for untyped objects. For programs that allocate objects of many different sizes we expect there to be some additional overhead for BDW because the free list will not be located in a register thus slowing down allocation. However the more significant effect is more subtle and once again has to do with cache behavior.

  Typically for BDW, the free list lies sequentially in memory, but distinct free lists are unrelated. A program that allocates a data structure containing objects of multiple sizes will have those objects scattered across multiple free lists with BDW. With MCC the objects will be laid out sequentially. This may lead to better locality for MCC if the access pattern is similar to the allocation pattern. This is typically the case when traversing recursive data structures such as trees which are encountered in several of these benchmarks.

- *Allocation.* The allocation sequence for MCC is seven instructions versus nine instructions for BDW. In addition MCC's allocation sequence requires only a single memory operation as opposed to four for BDW (see [25] for details). The advantages of faster allocation were hard to separate from the much larger cache effects without doing a cycle-level simulation.

- *Arrays.* Computing the array size is the only code other than allocation that is different for the two collectors. MCC requires two additional instructions to compute the array size. Only one benchmark (Checksum) was affected by this difference.

3.1.2 Collection Time

Figure 1 shows that most of the benchmarks spend much less time in the collector under MCC than under BDW, except for the smallest heap sizes. In the few cases that MCC's collector times are large relative to BDW's it is because the benchmark has a very deep stack (Knuth-Bendix), or there are untyped objects in the heap (Groebner). These factors are important because for MCC a quasi-pointer detection takes about 100 instructions whereas for BDW it takes only 30 instructions. At smaller heap sizes MCC performs more collections, further enhancing the relative cost of stack scanning and hence quasi-pointer detection.

We were not able to determine how useful type information was either in reducing the size of the live data or improving collector performance. Because these benchmarks, except for Checksum, manipulate small values that do not look like pointers we believe the effect on the size of the graph to be small.

3.1.3 Space

For all the benchmarks MCC requires more space than BDW. This is a combination of the space needed to copy objects, the extra overhead of the header word, and space lost to fragmentation. Because MCC initiates a collection whenever the heap becomes two-thirds full, the live data must fit in one-third of the heap. Since most of these benchmarks primarily allocate objects of size two (Figure 3) the live data for MCC is one and a half times larger than the live data for BDW. BDW can successfully collect the heap, ignoring the mark stack, whenever the live data fits in the heap. From these facts it follows that MCC can collect a benchmark as long as $\frac{9}{2}live < heap$. So MCC requires $4\frac{1}{2}$ times as much space as BDW. This is consistent with our empirical results. Sometimes less space is required because of the mark-stack for BDW, or differing object sizes. Although fragmentation could further increase the space requirements for MCC, it does not play a large role in these benchmarks.

Overall the running times for MCC are comparable with overall running times for BDW, being anywhere from 25% faster to 15% slower. In all cases, garbage collection times for MCC were better than for BDW and got progressively better as the heap size grew.

3.2 Collector Internals

MCC is a fairly well tuned implementation of our algorithm. The two optimizations that had the greatest impact were: specializing the copy loop and object dispatch for objects of size two, and inlining the inner loop of the collector (about 1000 lines of assembly code).

Figure 4 shows the breakdown of collector time into the various phases of the collector. Checksum and Life spend

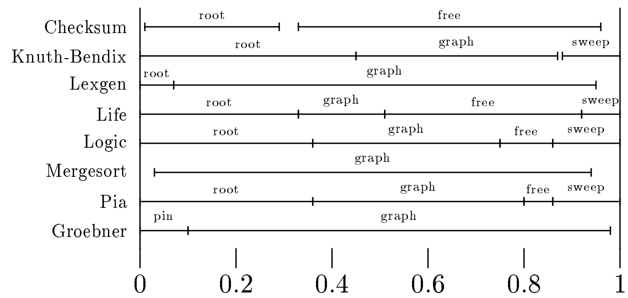


Figure 4: Percentage of collector time (MCC) spent: pinning objects (pin), scanning roots (root), graph traversal (graph), freeing pages (free), sweeping (sweep). (Values less than 5% have been omitted.)

Benchmark	Average per Collection			
	Roots	PP	Roots/PP	Frag.(KB)
Checksum	35	5	7	8
Knuth-Bendix	6126	917	7	1358
Lexgen	175	46	4	112
Life	50	7	8	11
Logic	234	47	5	67
Mergesort	36	10	4	7
Pia	325	50	6	78
Groebner	118	304	.4	585

Table 3: All data is taken at the canonical heap size (see Table 2). PP stands for promoted page. The Roots/PP column was measured for each collection and averaged, and is not necessarily the same as dividing column one by column two. Fragmentation measures the amount of unused space on promoted pages.

such a large proportion of collection time in stage *free* because they have small live sets and allocate a lot of data. For these two benchmarks copying the live data is cheaper than traversing the linked list of pages to free each one.

Most of the benchmarks spend a significant proportion of collector time in root scanning (traversing the registers, stack, and static area) because quasi-pointer detection is slow, stacks are fairly deep, and stack frames for the Sparc are relatively large due to register windows. Mergesort is an exception because it has a very shallow stack – about 11 stack frames. Groebner does not show root scanning as significant because it has so much live data, and many untyped objects that are very slow to scan.

The fragmentation column in Table 3 shows the amount of space on promoted pages that was invalidated by the sweep phase. Although it would be possible to reuse this space, MCC does not do so. As the table shows, for most of these benchmarks MCC wastes less than 100 KB.

We included the roots per promoted page because the data suggest a technique for speeding quasi-pointer detection. Since in most cases there are five or more quasi-pointers (roots) per promoted page, MCC could batch these quasi-pointer tests thus amortizing the cost of scanning the page over several detections.

3.3 Benchmarks

This section gives a detailed analysis of the individual benchmarks.

Checksum (Figure 1(a)) allocates many 2 and 3 word objects and one 4 KB array. Although Checksum allocates 1 GB of data only 46 objects are live. We expect copying collection to perform well in this setting.

Although collector times are much shorter for MCC than for BDW, client times are longer. We speculate that this is a result of cache behavior due to header words for MCC, and segregating objects based on size for BDW.

Checksum has another peculiarity that may account for some of the time. Although array sizes for this benchmark are constant, the compiler is unable to lift the array size calculation out of many of the loops. For BDW the array size is stored as the first element of the array, but for MCC the array size must be extracted from the header word – requiring an extra shift and an add.

MCC's collector times may be further enhanced because Checksum uses bit packing which may lead to some false quasi-pointer identifications by BDW, thus artificially increasing the size of the live graph.

Knuth-Bendix (Figure 1(b)) allocates many 1, 2, and 3 word objects. Because it uses many exception handlers, the compiler is unable to take advantage of tail recursion, so Knuth-Bendix generates a very deep stack (over 4000 stack frames).

The deep stack causes MCC to spend over half its time sweeping and scanning the stack. Since MCC uses a slower algorithm to infer quasi-pointers, and as many as 6000 quasi-pointers are found per collection, BDW and MCC collection times are comparable. Collection times for MCC do not drop uniformly because they are highly dependent on the depth of the stack when collections occured.

MCC requires four times more space than BDW. This is surprisingly low considering that 1 MB is lost to fragmentation at each collection. We surmise that the mark stack for BDW may be quite large.

Lexgen (Figure 1(c)) allocates only 75 MB. The speedup in the client time for Lexgen under MCC occurs because of cache behavior. Lexgen allocates an abstract syntax tree which contains nodes of many different sizes. Under MCC these are laid out in memory in the same order they are allocated. This happens to correspond to the access pattern of the program. For BDW, each node of a different size is in a different free list and therefore has no spatial locality. Other effects, such as the shorter allocation sequence for MCC may also play a role here.

Life (Figure 1(d)) allocates lists of pairs of integers. The pairs are allocated atomically for BDW and make up about 35% of the heap.

Client times are comparable under the two collectors although adding a "header" word slowed BDW's client time by 30%. Client times for MCC are faster because BDW maintains multiple free lists – one for atomic objects and one for untyped objects. Because Life sequentially examines each pair in the list, MCC gets much better locality despite the extra header word.

Logic (Figure 1(e)) allocates an abstract syntax tree containing nodes of various sizes. As shown in Figure 2 cache effects are minimal for this benchmark. The observed 12% speedup in client times under MCC is a consequence of a faster allocation sequence and the mix of object sizes which leads to better cache behavior for MCC.

Collection times are shorter for MCC because Logic has a shallow stack, and only about 30 KB of live data.

Mergesort (Figure 1(f)) allocates 330 MB of two-word list elements. At each collection about 160 KB are live. Because this benchmark only allocates elements of size 2, BDW need only maintain a single free list.

The client times under MCC are 20% slower because of locality. Although under both BDW and MCC the lists are arrayed in memory sequentially, the extra header word needed by MCC causes the list to consume one and half times as much memory, thereby reducing the effectiveness of the cache. The experimental data in Figure 2 confirms that locality effects are responsible for the observed slowdown.

Pia (Figure 1(g)) allocates many double-precision floating point values. The remainder of the data (45%) consists of closures (3 words) and list elements. Despite most of the data being allocated atomically MCC's collection times are better than BDW's in most cases.

The jump in collector time at 4 MB occurs because Pia's stack oscillates between a thousand and a hundred stack frames. Although MCC implements a heuristic to try collecting when the stack is shallow, there is not enough leeway for the heuristic to take effect until the heap reaches 8 MB.

Client times for MCC are comparable to times for BDW because locality is not important for this benchmark, and because the extra code needed to align the floating point values means that MCC's allocation routine is not shorter than BDW's in this case.

Groebner (Figure 1(h)) allocates only 2 word objects but has as much as 4 MB of live data (6 MB when we take header words into account). Groebner uses polymorphism that the TIL compiler is unable to eliminate at compile time. Because we do not yet dynamically generate header words, Groebner has untyped objects in the heap (about 300 per collection).

Because MCC's quasi-pointer detection is slow relative to BDW's, collection times are approximately the same under both BDW and MCC. Further work optimizing quasi-pointer detection may make a significant difference for this benchmark.

The slowdown in client time for MCC is due to locality effects as shown in Figure 2.

4 Future Work

The preliminary results presented above suggest several directions future research might take. It would be interesting to use a cycle-level simulator to pinpoint the performance effects we have observed, and to investigate the impact of cache architecture. Larger and longer-running benchmarks could also help determine whether these cache effects are important for a larger class of programs.

Further work is needed to fully tune MCC. In particular more sophisticated quasi-pointer detection schemes are possible. We are also exploring the possibility of storing the header words in a separate space. This will slow allocation and possibly collection, but may recover the cache behavior lost with the current approach.

Another direction of interest is to look at how the language context affects collector performance. For example, Java implementations use a header word on objects to point to the class. This header word can also be leveraged by the collector, thereby avoiding some of our overheads when compared to BDW.

We speculate that in a concurrent setting, a collector architecture similar to MCC's will be favored because each thread can allocate objects on a private page without synchronization. Only page allocation and garbage collection would require synchronization.

An intriguing possibility is that a compiler using accurate collection may benefit from a conservative collector. Traditional compilers using accurate collection generate additional code to establish *GC-safe* points where type information has been collected for all the registers, and stack. A conservative collector makes less stringent demands on the client and therefore may be able to recover gracefully when type information is unavailable, thus making GC-safe points unnecessary. Furthermore, linking against legacy code or external libraries may be facilitated if complete type information is not required.

5 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 BDW, and have worse cache behavior. In addition, MCC implements a slow pointer testing routine which penalizes it further in the presence of a deep stack or many conservative objects relative to BDW. Despite these drawbacks, MCC is comparable if not better than BDW on almost all benchmarks. The study in [15] suggests that more sophisticated techniques 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.

6 Acknowledgments

We extend our thanks to the many people who have provided valuable comments and suggestions: Giuseppe Attardi, Joel F. Bartlett, Hans J. Boehm, Chris Hawblitzel, Simon Peyton Jones, David Walker, Stephanie Weirich, and Steve Zdancewic.

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] 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.

[4] 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, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[5] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, Feb. 1988.

[6] J. F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical report, DEC WRL, Oct. 1989.

[7] 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.

[8] H.-J. Boehm. Space-efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, June 1993.

[9] H.-J. Boehm and D. R. Chase. A proposal for garbage-collector-safe C compilation. *C Language Translation*, 1992.

[10] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[11] 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.

[12] A. Diwan, D. Tarditi, and E. Moss. Measuring subsystem performance of programs using copying garbage collection. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, Portland Oregon, 1994.

[13] A. Diwan, D. Tarditi, and E. Moss. Memory-system performance of programs with intensive heap allocation. *Transactions on Computer Systems*, 13(3):244–273, Aug. 1995.

[14] D. Edelson. A mark-and-sweep collector for c++. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 51–58, New Mexico, Jan. 1992.

[15] 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.

[16] 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.

[17] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995.

[18] 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.

[19] Objective caml. `http://pauillac.inria.fr/ocaml/`.

[20] 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.

[21] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, 1989.

[22] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. *Lecture Notes in Computer Science*, 983:366–81, 1995.

[23] Z. Shao and A. W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, pages 150–161, Orlando, June 1994.

[24] 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.

[25] F. Smith and G. Morrisett. Mostly copying collection: A viable alternative to conservative mark-sweep. Technical report, Cornell, 1997.

[26] 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.

[27] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Nov. 90.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

	Pin	Root	Graph	Free	Sweep
Checksum	1	28	4	63	4
Knuth-Bendix	0	45	42	1	12
Lexgen	0	7	88	2	3
Life	0	33	18	41	8
Logic	0	36	39	11	14
Mergesort	0	3	91	4	2
Pia	0	36	44	6	14
Groebner	10	0	88	1	1

Table 4: Percentage of collector time spent in each phase. Data corresponding to Figure 4.

[32] P. R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [31]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

[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 256 MB UltraSparc-5 running Solaris 2.6. The machine was lightly used while we ran our experiments.

We present here the numbers used to obtain the graphs in the paper with an explanation for each.

The data in Figure 4 was obtained by wrapping timers around each of the distinct phases of the collector and running each benchmark at the canonical heap size listed in Table 2. The actual numbers are shown in Table 4.

The data in Figure 3 were obtained by instrumenting the allocation macros to record the number of objects allocated of each size from 1 to 3, and greater than 3 (4+). We also instrumented the atomic allocation routines for BDW to count the number of objects allocated of each size. From this information we computed the percentage of the data that were allocated atomically. Table 5 gives the actual numbers.

The data in Figure 2 was obtain by adding an extra header word to every inlined allocation for BDW. We compared the client times for BDW with and without the header word and computed the percent slowdown using BDW without the header word as a baseline. All runs were made at the canonical heap size for each benchmark. Table 6 gives the actual percentages.

Table 7 contains the timing data corresponding to Figure 1. These numbers were obtained by running each benchmark at each heap size ten times, throwing out the worst run, and averaging. We used high-resolution timers with overhead of about one microsecond to take these measurements. In these tables we also include the data for the best overall running times. In the text we chose to use the average because at small heap sizes the running time can be sensitive to the initial state of the stack and registers. When no data is listed in the table that is because that benchmark could not be run at that heap size.

	1	2	3	4+	Atomic
Checksum	0	50	50	0	0
Knuth-Bendix	31	38	31	0	2
Lexgen	0	87	4	8	26
Life	0	100	0	0	37
Logic	5	62	31	2	0
Mergesort	0	100	0	0	0
Pia	2	80	17	1	53
Groebner	0	100	0	0	1

Table 5: Percentage of objects allocated at each size, and the percentage of data allocated atomically for BDW. Corresponds to the data shown in Figure 3.

	MCC	$BDW+1$
Checksum	23.6	11
Knuth-Bendix	0.3	2.6
Lexgen	-15.0	8.1
Life	-0.5	31.6
Logic	-12.1	0.6
Mergesort	21.7	20.9
Pia	2.0	5.9
Groebner	16.3	14.4

Table 6: Percent slowdown for BDW with an extra header word and MCC client times relative to a baseline of BDW. Corresponds to the data shown in Figure 2.

		Client		GC		Client		GC		Client		GC	
		B	A	B	A	B	A	B	A	B	A	B	A
		Checksum				Knuth-Bendix				Lexgen			
1	BDW	15.70	15.77	11.72	11.72								
	MCC	19.31	19.37	1.67	1.69								
2	BDW	15.70	15.73	10.79	10.87					18.50	18.88	1.94	1.95
	MCC	19.27	19.32	1.35	1.35								
4	BDW	15.77	15.75	10.32	10.37	18.10	18.16	2.29	2.29	18.94	19.20	1.33	1.34
	MCC	19.33	19.35	1.22	1.24					15.17	15.47	1.06	1.09
8	BDW	15.66	15.72	10.31	10.35	18.15	18.19	1.84	1.85	19.10	19.34	1.20	1.19
	MCC	19.36	19.43	1.14	1.16					16.16	16.44	0.56	0.59
12	BDW	15.66	15.70	10.34	10.35	18.12	18.18	1.80	1.80	18.84	19.07	1.18	1.18
	MCC	19.41	19.47	1.12	1.12					16.47	16.73	0.40	0.40
16	BDW	15.68	15.69	10.32	10.37	18.15	18.24	1.57	1.56	19.00	19.11	1.26	1.26
	MCC	19.45	19.51	1.11	1.11	18.25	18.30	2.33	2.42	16.68	16.81	0.34	0.36
24	BDW	15.68	15.72	10.52	10.54	18.41	18.45	1.73	1.75	18.96	19.00	1.32	1.36
	MCC	19.54	19.60	1.09	1.11	18.28	18.34	0.62	0.62	16.89	17.07	0.21	0.22
32	BDW	15.79	15.81	10.57	10.64	18.59	18.64	1.71	1.72	18.96	19.03	1.45	1.47
	MCC	19.60	19.66	1.10	1.09	18.40	18.49	1.46	1.45	17.00	17.18	0.17	0.17
48	BDW	15.72	15.79	10.88	10.89	19.04	19.08	2.01	2.05	18.90	19.06	1.71	1.74
	MCC	19.77	19.83	1.08	1.09	18.61	18.66	0.50	0.50	17.50	17.62	0.10	0.10
64	BDW	15.73	15.79	11.11	11.16	19.49	19.57	2.21	2.24	18.85	18.96	2.04	2.03
	MCC	20.06	20.14	1.10	1.09	18.86	18.93	0.65	0.64	17.83	17.95	0.15	0.16
		Life				Logic				Mergesort			
1	BDW	11.22	11.28	2.92	2.93	6.76	6.77	2.54	2.54	5.55	5.55	20.04	20.19
	MCC	11.05	11.10	0.58	0.58	5.81	5.84	1.46	1.47				
2	BDW	11.15	11.18	2.73	2.75	6.77	6.80	2.28	2.29	5.51	5.56	7.99	8.10
	MCC	11.05	11.12	0.42	0.42	5.89	5.90	0.76	0.78				
4	BDW	11.22	11.28	2.67	2.66	6.77	6.82	2.19	2.20	5.53	5.53	5.55	5.62
	MCC	11.11	11.16	0.32	0.31	5.88	5.90	0.46	0.46	7.06	7.22	4.67	5.62
8	BDW	11.20	11.26	2.69	2.70	6.78	6.79	2.21	2.22	5.54	5.53	4.35	4.38
	MCC	11.16	11.21	0.25	0.25	5.96	5.97	0.31	0.31	6.69	6.73	1.43	1.44
12	BDW	11.32	11.34	2.75	2.76	6.79	6.81	2.26	2.29	5.53	5.53	4.05	4.08
	MCC	11.18	11.24	0.23	0.23	6.01	6.06	0.26	0.26	6.90	6.93	1.54	1.55
16	BDW	11.23	11.30	2.81	2.81	6.80	6.83	2.33	2.35	5.49	5.53	4.03	4.02
	MCC	11.26	11.27	0.22	0.22	6.08	6.11	0.23	0.23	6.95	6.99	1.43	1.44
24	BDW	11.21	11.27	2.92	2.95	6.83	6.85	2.46	2.47	5.53	5.54	4.02	4.04
	MCC	11.34	11.40	0.22	0.22	6.18	6.21	0.20	0.20	6.98	7.01	1.04	1.06
32	BDW	11.38	11.46	3.01	3.01	6.84	6.84	2.58	2.59	5.53	5.61	4.03	4.03
	MCC	11.41	11.47	0.22	0.22	6.30	6.33	0.19	0.19	6.99	7.04	0.80	0.82
48	BDW	11.24	11.30	3.27	3.31	6.90	6.94	2.83	2.87	5.56	5.59	4.14	4.16
	MCC	11.60	11.62	0.21	0.21	6.54	6.56	0.17	0.17	7.23	7.29	0.65	0.65
64	BDW	11.27	11.33	3.46	3.49	6.92	6.93	3.05	3.10	5.52	5.54	4.40	4.43
	MCC	11.81	11.88	0.21	0.21	6.83	6.87	0.17	0.18	7.38	7.44	0.51	0.52
		Pia				Groebner							
1	BDW	11.04	11.09	3.11	3.12								
	MCC												
2	BDW	10.95	11.04	2.72	2.73								
	MCC	11.09	11.14	1.44	1.52								
4	BDW	10.97	11.32	2.58	2.61								
	MCC	10.96	11.19	2.28	2.27								
8	BDW	10.93	11.00	2.58	2.58								
	MCC	11.03	11.21	0.71	0.72								
12	BDW	10.97	11.00	2.64	2.63								
	MCC	11.10	11.28	0.59	0.59								
16	BDW	10.96	11.02	2.67	2.67	35.38	35.46	21.84	21.79				
	MCC	11.10	11.26	0.43	0.43								
24	BDW	10.97	11.75	2.80	3.03	35.29	35.40	17.49	17.50				
	MCC	11.23	11.36	0.38	0.38	41.37	41.40	24.65	24.63				
32	BDW	10.96	11.01	2.86	2.88	35.31	35.48	15.79	15.82				
	MCC	11.32	11.41	0.27	0.27	41.24	41.28	18.96	19.14				
48	BDW	10.96	11.06	3.14	3.17	35.58	35.65	14.61	14.66				
	MCC	11.47	11.63	0.25	0.25	41.75	41.80	12.81	12.97				
64	BDW	11.13	11.18	3.40	3.44	35.70	35.81	14.04	14.10				
	MCC	11.68	11.85	0.24	0.24	41.62	41.64	10.02	10.11				

Table 7: Timing data for the benchmarks as depicted in the text. The column B contains the data for the run with the best overall time, and column A shows the average as plotted in Figure 1