Appeared in Software: Practice and Experience 22 (8), 659-672, Aug. 1992.

Garbage Collection Alternatives for Icon

Mary F. Fernandez and David R. Hanson
Department of Computer Science, Princeton University,
Princeton, NJ 08544 USA

June 1991, Revised March 1992

Abstract

Copying garbage collectors are becoming the collectors of choice for very high-level languages and for functional and object-oriented languages. Copying collectors are particularly efficient for large storage regions because their execution time is proportional only to the amount of accessible data, and they identify and compact this data in one pass. In contrast, mark-and-sweep collectors execute in time proportional to the memory size and compacting collectors require another pass to compact accessible data. The performance of existing systems with old compacting mark-and-sweep collectors might be improved by replacing their collectors with copying collectors. This paper explores this possibility by describing the results of replacing the compacting mark-and-sweep collector in the Icon programming language with four alternative collectors, three of which are copying collectors. Copying collectors do indeed run faster than the original collector, but at a significant cost in space. An improved variant of the compacting mark-and-sweep collector ran even faster and used little additional space.

KEYWORDS: storage management, garbage collection, high-level languages, Icon.

Introduction

Automatic reclamation of inaccessible memory — garbage collection — has long been an important aspect of very high-level languages, such as Lisp and SNOBOL4. Garbage collection is emerging as an essential component of a wide range of modern programming language systems. Examples include very high-level languages, such as Icon [1], object-oriented languages with late binding times, such as Smalltalk [2] and Self [3], and new languages with traditional compile-time type systems such as ML [4], Eiffel [5], Oberon [6], and Modula-3 [7]. Garbage collectors have also been implemented for languages that were not originally designed to support garbage collection, such as Modula-2+ [8], C++ [9], and even C [10].

Recent implementations tend to use *copying* collection algorithms instead of *mark-and-sweep* algorithms [11, 12]. Copying algorithms take time that is proportional to the amount of accessible data and identify and compact accessible data in one pass. Mark-and-sweep algorithms take time that is proportional to the amount of accessible *and* inaccessible data and the compacting variants require another pass to compact accessible data. Copying collectors use more memory because they require two separate spaces, but they tend to improve locality of reference because they place objects near their referents. Large storage regions (e.g., tens of megabytes) may amplify the advantages of copying collectors [13].

Recent advances suggest that existing systems with "old" compacting mark-and-sweep collectors might benefit from new collectors. Icon [1] is a prime example. The key question is whether or not a new collector can yield significant performance improvements for most Icon programs. The remainder of this paper describes the results of implementing several new compacting collectors for version 8 of Icon [14].

Documented experiences that might help choose a collector for a specific system are scarce. Those that are available are necessarily system specific; it is often difficult to translate results from one system to another, especially if the systems differ significantly. Significant differences can exist even for similar

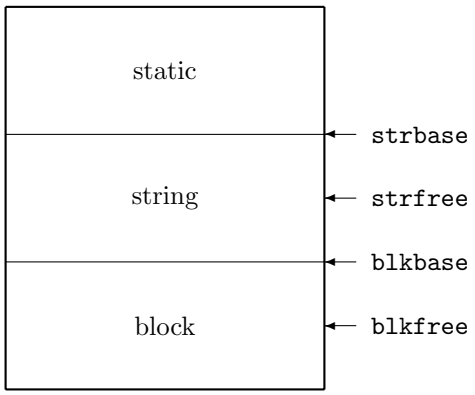

Figure 1: Memory Layout.

languages. For instance, recent reports on Lisp systems [15, 16] are likely to be valuable for other systems in which memory is composed of small, fixed sized, homogeneous objects, but those results may be less applicable when memory holds a wide variety of variable sized, heterogeneous objects, as in Icon. Also, most systems have complicating idiosyncrasies, such as the preponderance of variable length strings in Icon.

The complexities of specific systems make measurement results difficult to interpret and to apply. Few real systems are documented well enough to understand fully the ramifications of their designs and implementations and how they might be reflected in other systems. Icon's implementation is well documented [17], and its source code is available publicly. The results reported here are, of course, specific to Icon, but Reference [17] and the source code provide a context in which to evaluate the applicability of these results to other systems.

Icon

Icon is a very high-level imperative language with a rich repertoire of facilities for string and structure processing [1]. It is available on a wide range of computers from personal to supercomputers and it is widely used; over 10,000 copies have been distributed.

In Icon, values are typed, not variables. Built-in data types include numerics, character sets, strings, sets, lists, associative tables, and records. The latter four aggregate types can hold values of any type. Numerics, character sets, and strings are atomic values; operations on them produce new values. Aggregates use pointer semantics; operations on them can change existing values as well as produce new ones. Strings and aggregates can be of arbitrary sizes, and these sizes can change during execution. Memory management is automatic.

During execution, storage is divided into the three regions depicted in Figure 1. Values that cannot be moved, such as I/O buffers and execution stacks, are allocated in the static region. These values are fixed sized, system dependent, and are never reclaimed. Thus, garbage collection alternatives do not involve this region.

Strings are allocated from the string region. Values in Icon are represented by two-word *descriptors*, which contain a type code and other type-specific data, e.g., the value of an integer. For strings, these type-specific data are the length of the string and the location of its first character in the string region. The string region contains only string data, so allocation is fast: `strfree` is simply incremented by the requested amount. This representation simplifies many string operations. For example, if `s` has been assigned the string `"hippotamus"`, the substring `"pot"` can be formed in constant time by returning a descriptor with a location equal to the location of `s` plus 3 and a length of 3. Likewise, concatenation to a

newly created string can omit copying its left operand if it ends at `strfree`, and sometimes the operands of concatenation are already adjacent, so concatenation is trivial. Such considerations are particularly important to the efficient implementation of string scanning — Icon's "pattern matching" [18].

All other values are allocated in the block region. The type-specific data in descriptors for character sets and aggregates point to *blocks* in the block region. Blocks are analogous to nodes or objects in other systems and have type-specific sizes and layouts and most hold one or more descriptors [17]. As in the string region, allocation is trivial: `blkfree` is incremented by the size of the requested block.

Garbage collection occurs when a request for space cannot be satisfied and is described fully in Chapter 11 of Reference [17]. Briefly, collection begins with a marking phase that locates all blocks and strings accessible from a *root* set, which includes values in the static region, the stack, global variables, and several internal variables. As accessible strings are located, pointers to their descriptors are appended to the *qualifier list*, which is used during compaction. Space for this list begins at `blkfree`. Accessible blocks are marked by processing each block recursively. Each block has a header word that usually contains a block code, but for marked blocks, heads a list of descriptors that point to the block. This list is threaded through the descriptors themselves and is terminated by the block code in the last descriptor.

After marking, accessible strings in the string region are compacted by sorting the qualifier list by location and making a pass over the list identifying and moving accessible characters. This scheme takes into account the possibility of "overlapping" strings and preserves substrings. This pass also updates the locations in the descriptors pointed to from the qualifier list to reflect the new locations of the strings.

Next, two passes over the block region are made. The first pass computes the new locations for accessible blocks, which are identified by the presence of a list of descriptors in their headers, and, for each such block, traverses this list changing the descriptors to point to the block's new location. The block codes are also returned to the headers along with a mark. The second pass compacts accessible blocks, identified by header marks, and clears these marks.

If necessary, the string region is expanded by relocating the entire block region. This relocation is accomplished by collecting the block region as usual, but including the amount of expansion when computing new block locations. The entire — now compacted — block region is then shifted up. This possibility of expansion is why the two passes over the block region mentioned above cannot be combined. The qualifier list can also overflow the block region; if it does, which is rare, the block region is expanded by requesting more memory from the operating system.

Observations

Garbage collection can have a measurable effect on total performance. It accounts for 5–78% of total execution time for the programs in the test suite described below. This suite was used to understand the behavior of the existing collector and to guide the design of alternatives, described below. The measurements of the existing collector corroborate earlier work [19].

The maximum amount of accessible memory used during execution ranged from 200KB to 2MB for the test suite. These sizes are much smaller than the accessible data sizes in test suites used in comparative analyses in Lisp, for example, where sizes from 5–100MB are typical. But a few megabytes or less are typical of Icon programs on workstations, and even smaller sizes are typical on personal computers, such as the MacIntosh. Copying collectors that excel for large amounts of accessible data may not do so for smaller amounts.

Long-lived data is data that survives many collections; researchers have long recognized the importance of handling such data efficiently [20, 21, 22]. For the Icon test suite, 30–50% of the allocated data remains accessible to the end of execution occupying space that cannot be reclaimed. The existing collector does not move data unnecessarily, but a non-generational copying collector will move such data at each collection.

The existing memory management scheme caters to strings, but programs that do extensive string manipulation pay for it; for those programs in the test suite, 16–58% of collection time is spent constructing and sorting the qualifier list and compacting the string region. These programs would benefit

from alternatives that eliminate the qualifier list.

Dividing memory into two equal-sized regions wastes memory for programs that use mostly strings or mostly aggregates. This division complicates region expansion as described above. For example, executing the 63 small programs in the Icon program library [23] with their small test inputs generates 9,816 strings with a mean length of only 7.23 characters and a median length of 2. Strings longer than 100 characters were counted as 100-character strings, and only 2% of the 9,816 strings exceeded 100 characters. These data suggest that it might be equally effective to store strings in blocks and dispense with the separate string region.

Alternatives

The observations described above motivated the design and implementation of four alternative collectors for Icon.

An initial premise was that a copying collector might outperform the existing mark-and-sweep collector, so the first alternative, `copier`, is a simple copying collector for the block region. Simple copying collectors are rarely used alone; they are usually used in a generational collector [4]. `copier` serves only as a baseline for comparing other copying alternatives.

As in all copying collectors, the block region is divided into two semi-spaces. Allocation proceeds as in the existing collector from "old" space until a request cannot be satisfied. During collection, accessible data is copied from old space to "new" space, which also compacts the data, the roles of the spaces are reversed, and execution continues [11]. When a block is copied, a forwarding pointer is left in the original so that other descriptors pointing to the block can be re-aimed at its location in new space.

The second alternative, `string`, eliminates the separate string region and allocates strings and blocks in a single region and eliminates the qualifier list, which reduces sharing after collection as described below. `string` allocates a 4KB "string block" and doles out space for strings from this block. When it becomes full, another string block is allocated. Collection proceeds as in `copier`. When an accessible string is located, it is appended to the "current" 4KB string block in new space, creating one if necessary. While this scheme eliminates the qualifier list and its expensive processing, its space cost can be high because it duplicates strings that share characters before collection. For example, suppose N accessible string descriptors point to an M-byte string block. Collection might create N strings totaling as much as $N \times M$ bytes. Excessive expansion would suggest significant sharing, which should be highest for programs that create many substrings.

The third alternative, `string2`, is similar to `string`, but avoids its worst case behavior. As blocks are copied to new space, accessible string descriptors are added to a qualifier list as in `copier`, but the list is never sorted, and string blocks are *not* copied. Instead, string block headers record "low" and "high" water marks, which give the lowest and highest addresses, respectively, of accessible string data within the block [21, 24]. After copying all other blocks to new space, the data between the low and high water marks in each string block in old space are copied into 4KB string blocks in new space as in `string`, and the qualifier list is scanned to update the string descriptors. The qualifier list is at the end of the region and is expanded, if necessary. `string2` is otherwise identical to `string`. Note that `string2` saves all characters between the low and high water marks, even if they are inaccessible.

The last alternative, `mark&compact3`, is a single-region variant of the original mark-and-compact algorithm that handles strings as in `string2`. The marking phase builds lists of descriptors that reference accessible blocks as in the original algorithm, adds strings to a qualifier list as in `string2`, and computes `string2`'s low and high water marks for string blocks. The next phase adjusts descriptors as in the original algorithm, but the low water mark is taken into account in adjusting string descriptors, and both the low and high water marks are used to compute the new size and location of a string block. The final phase compacts accessible blocks as in the original algorithm, but copies only the data between the low and high water marks in string blocks. As in the original algorithm, `mark&compact3` does not copy long-lived data unnecessarily and does not incur the space cost of two semi-spaces. `mark&compact3` is similar to SITBOL's

test program	length in lines	accessible blocks after collection	accessible strings after collection
`best`	8		
`worst`	8		
`string0`	12		
`string50`	12		
`callgraph`	54	51–78%	2– 7%
`pslist`	426	24–78	1– 3
`burg`	625	72–79	4– 5
`typsum`	2804	56–80	3– 8
`mkgen`	991	32–76	4–36
`concord`	53	56–78	2–19

Table 1: Test Suite

collector [21].

The original algorithm and the algorithms described above expand regions after collection, if necessary, in order to avoid collecting too frequently. For example, if a collection yields only a small amount of free space, another collection is imminent. Expanding regions by 25% avoids excessive collecting. For the copying collectors, both semi-spaces are expanded, which increases their space cost.

Measurement Results

Test Suite

The test suite consists of the ten programs summarized in Table 1, which, for each program, gives its length and the percentage of accessible block and string data when run with the original collector. These percentages show the range of the amount of accessible after each collection, e.g., at each collection in `callgraph`, 51–78% of bytes in the block region were in accessible blocks. For all of the test programs, later collections tended to have the higher percentages.

The first four programs listed in Table 1 are artificial programs designed to expose the bounds of expected improvements for each alternative. `best` and `worst` characterize, respectively, the best and worst programs for a copying collector (and vice versa for mark-and-sweep collectors). `best` generates almost all garbage:

```
procedure main()
    local t, i
    t := table();
    every i := 1 to 500000 do t[i]
end
```

This program builds a table of 500,000 entries by referencing each entry, which allocates space, but each entry is inaccessible because it is never assigned a value. `worst` is similar except that it does 100,000 assignments `t[i] := i` instead of just referencing `t[i]`, which allocates only accessible entries and hence creates *no* garbage.

`string0` and `string50` are similar. `string0` creates 500,000 strings of random lengths between 1 and 100 characters and hence creates only string garbage. `string50` creates 75,000 random-length strings and assigns them to the entries in `t` with probability one-half, i.e., approximately 50% of the entries.

The other six programs listed in Table 1 are real programs provided by Icon users. They vary in size, execution time and number of collections, but most do extensive string manipulation as do most Icon programs.

`callgraph` reads compiler-generated assembly language files, computes the call graph, and prints an indented representation of the graph and a procedure index. The sample input for `callgraph` is the assembly code generated from Icon's run-time system, 22,743 lines of C; it references 334 procedures and has 1558 call-graph edges.

`pslist` reads C, Fortran, or Ratfor source files and generates PostScript that prints listings with cross-reference indices. Unlike `callgraph`, which generates its output after reading all of its input, `pslist` generates much of its output as it executes.

`mkgen`, a large program by Icon standards, reads a compact code-generator specification and emits a code generator in C [25]. `mkgen` is used to generate the code generators for `lcc` [26]. The input is the VAX specification.

`burg` is similar in function to `mkgen`, and its input is a VAX specification.

`typsum` reads "ucode," Icon's intermediate representation [17], and performs type inference. Its minimal output summarizes the results of type inference, e.g., number of variables with no type, etc. A refined variant of `typsum` is part of the new Icon compiler [27].

`concord` is a concordance program from the Icon programming library. It produces an index of the words in its input by building a table indexed by words and containing lists of line numbers. It prints a line-numbered copy of its input, and, at the end of the program, the table and each list are sorted and the line numbers are concatenated and printed. The input to `concord` is the text of *Macbeth*.

Results

Data was collected by running the test suite with each of the alternative collectors described above. In each case, execution began with 130KB regions, divided into two 65KB semi-spaces for the copying collectors. As mentioned above, regions or semi-spaces are expanded if there is not enough free space to satisfy the request that triggered the collection. Specifically, a region is expanded so that the resulting free space is twice the amount of the triggering request or 25% of the region's size, whichever is larger.

All times reported are the average elapsed times in seconds on a DECStation 5000, averaged over at least 4 runs. All reported runs achieved at least 90% utilization (i.e., the ratio of times $(user + system)/elapsed \geq 0.90$).[1] The raw data includes elapsed time, garbage collection time, number of collections, and the maximum size of the storage region. These data appear in the Appendix and are summarized in the figures below. All data in the figures are normalized so that the original collector runs in 100 time units, i.e., they display $100 \times X/T$ where T is the execution time of each test program using the original collector, and X is the execution time using alternative X.

Space costs are reported similarly as X/S where S is the maximum storage size of each test program using the original collector, and X is the maximum storage size using alternative X. If storage for a program never exceeded 130KB, X is reported as 130KB.

Figures 2 and 3 show the execution times for the original and for each alternative algorithm. The number of collections appears above each bar, and the black line in each bar indicates the portion of the execution time spent in collection, i.e., the black lines are positioned at $100 \times C/T$, where C is the collection time.

The results for the artificial programs follow the expected trends, e.g., the copying collectors (`copier`, `string`, and `string2`) do poorly on `worst` because it generates no garbage, and they do well on `best` and `string50` because `best` generates only garbage and `string50` generates 50% garbage. `string`, `string2`, and `mark&compact3` do not have separate string regions, so their storage regions are twice as large as `copier`'s. Consequently, they do fewer collections and thus do better than `copier` on `best`. `mark&compact3`'s performance on `best` is lower than that of `string` and `string2` because, being a mark-and-sweep collector, it must scan all of the garbage, which is most of memory for `best`.

[1] The iteration counts for `best`, `worst`, `string0`, and `string50` were chosen to yield this high utilization.

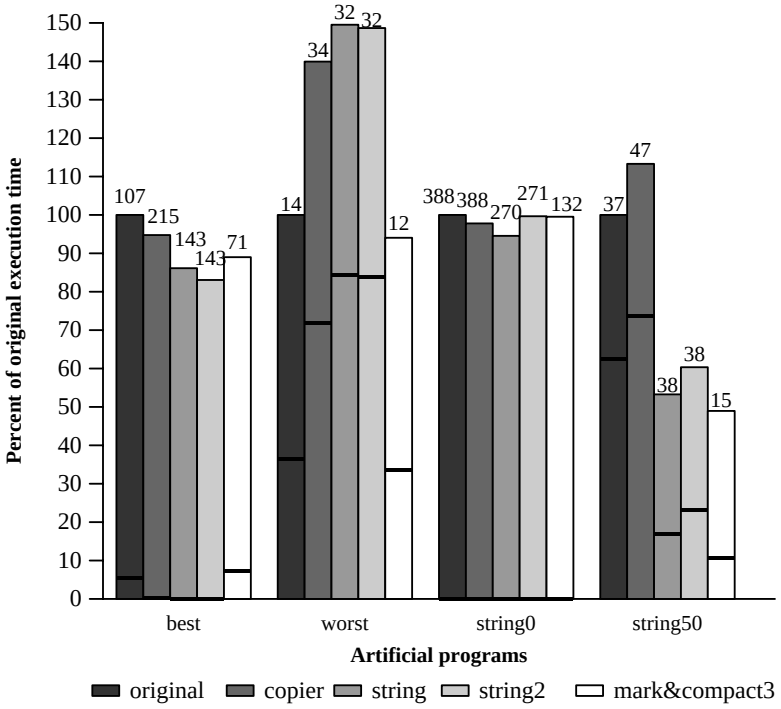

Figure 2: Reduction in Execution Time

`copier` does poorly on `string50` because `copier` uses the original collector for strings and repeatedly copies the long-lived data in the block region. Using `mark&compact3` on `worst` shows a slight improvement because almost all of the data is long-lived and it avoids copying this data. It does well on `string50` for the same reason.

The small difference in performance between `string` and `string2` and `mark&compact3` on `string0` is due entirely to the different string representation used in the latter two variants.

Three of the alternative collectors reduce execution time for the real programs and some reduce it dramatically. Figure 3 shows the importance of collecting strings efficiently: `copier`, which uses the original collector for strings, performs respectfully only for `burg` and `typsum`, which do less string manipulation than the other test programs. For instance, most of `concord`'s 109 collections are because the string region is full. `mark&compact3` is competitive with the `string` and `string2` copying collectors and is often superior. `string` and `string2` collect strings efficiently, but their performance can suffer when most strings are long-lived as in `callgraph` and `pslist`.

The reductions in execution time come at a significant cost in space, however. Figures 4 and 5 display the space costs for each alternative, as described above.

The copying alternatives pay for the second semi-space; at any time, only one space contains accessible data, so these alternatives can use twice as much memory as the maximum amount of accessible data. The space costs for `string` include the effects of string duplication described above. Space costs above 2 can be attributed to this effect, which, as Figure 5 shows, is minimal. `string2`'s space cost is often higher than `string`'s because it constructs a qualifier list and saves some characters that are inaccessible. This latter effect is particularly noticeable in `string50`: every other string is garbage, so almost one-half of every string block is tied up with inaccessible data.

For most of the test programs, `mark&compact3` uses little more space than the original collector. `mark&compact3` has the lowest space cost because it does not require an unused semi-space. `mark&compact3` uses slightly less space than the original collector for `worst` because it uses most of its storage for blocks; the original pays for its initial 130KB string space, most of which goes unused for `worst`.

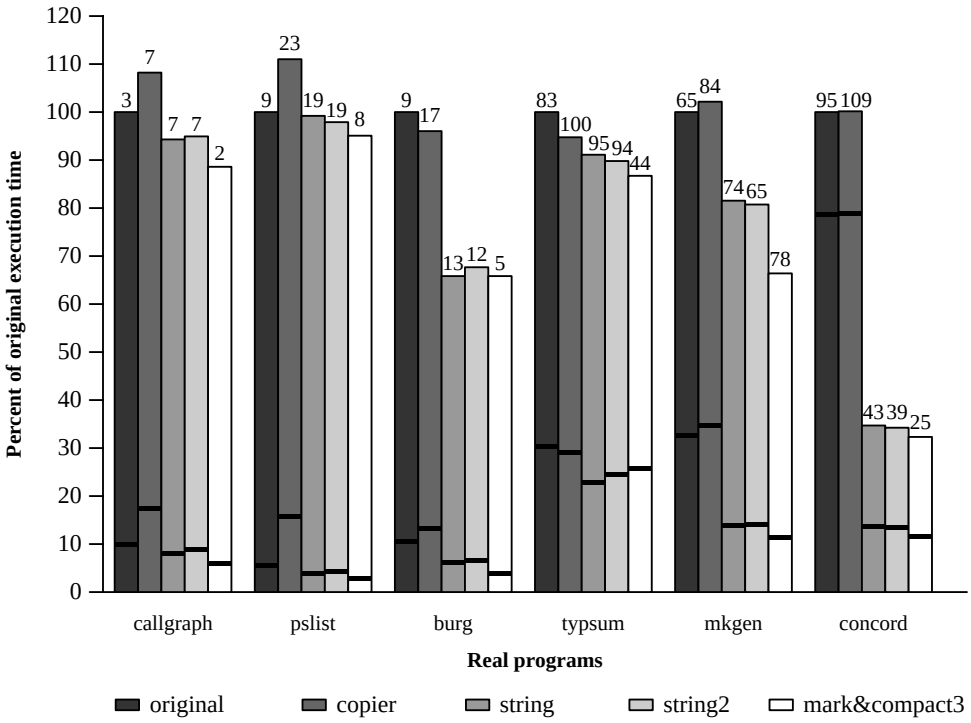

Figure 3: Reduction in Execution Time

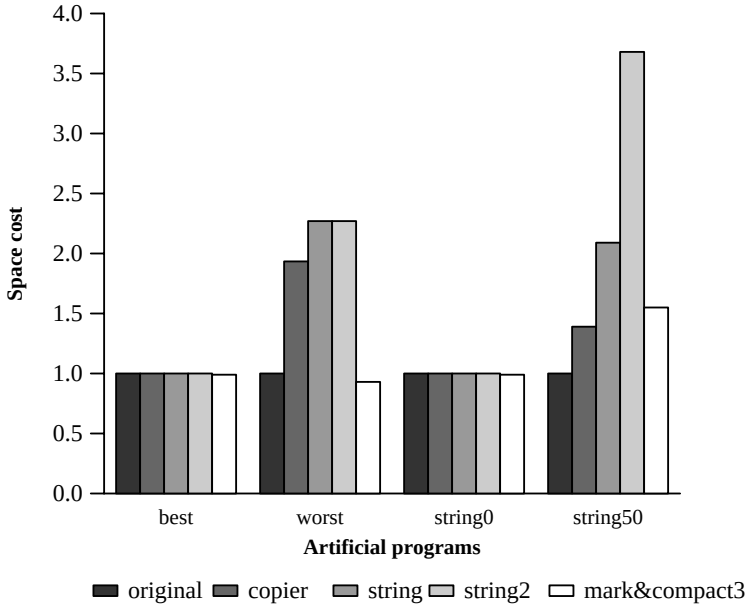

Figure 4: Space Costs

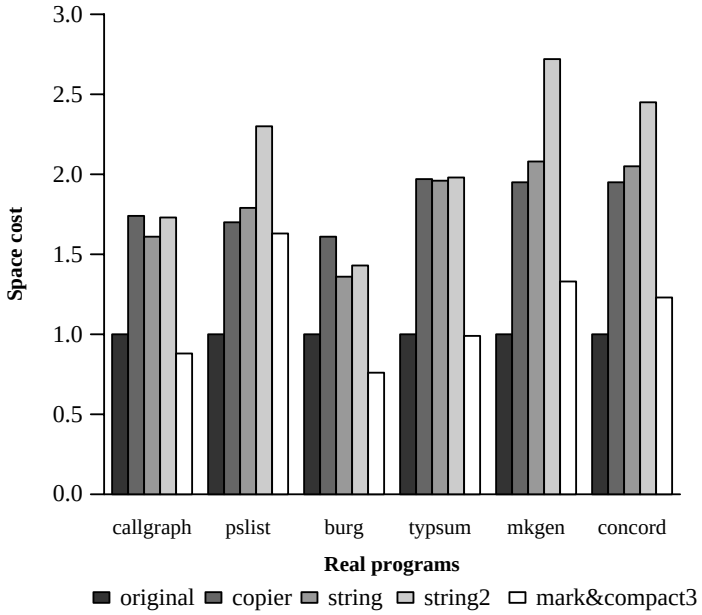

Figure 5: Space Costs

Generational Collection

Generational collectors [22] are another alternative that should be explored. These collectors perform best for languages in which most of the inter-object pointers are from recently allocated "new" objects to previously allocated "old" objects, i.e., languages in which old objects are rarely changed. Generational collectors work well for mostly applicative languages like ML [4] and for some very high-level imperative languages like Smalltalk [28], which is similar to Icon in some respects (e.g., run-time typing, heterogeneous structures).

A generational collector was not implemented for two reasons. Icon's original collector is similar to the one used in SITBOL [21], an implementation of SNOBOL4. SITBOL's collector used a scheme similar to that used in generational collectors. It remembered the value of `blkfree` after a collection and treated all blocks below this value as part of the root set at the next collection. These blocks are analogous to the older generation in a generational collector with two generations. The intent was to reduce collection time by not processing long-lived blocks, but measurements revealed that this scheme saved only 5% [24]. Icon and SNOBOL4 are very similar, both as languages and in their implementation techniques, so this measurement suggests the improvement from a generational collector might be small.

The more important reason is that the current implementation of Icon's goal-directed evaluation mechanism involves numerous assignments buried in the run-time system [17]. Generational collectors must maintain "remembered sets" — lists of old blocks that point to newer blocks. Maintaining these sets, which might become large, would require massive changes to Icon's run-time system and compiler. The result would be incomparable to the original implementation.

Moreover, simply changing these assignments is the wrong way to test a generational collector for Icon. Icon's implementation should be redesigned to use representations and techniques that best suit generational collectors, e.g., using virtual memory hardware for detecting changes to old objects [11]. Given Icon's wide use, this kind of study is undoubtedly worthwhile, but is well beyond of the scope of replacing just the garbage collector.

Discussion

As the measurements detailed above demonstrate, the alternative collectors usually made the test suite's real programs run faster by a few tens of percent. But some alternatives achieve this improvement at a significant cost in additional space. For example, `string` and `string2` often run faster than the original, but they have a high space cost. The execution times for `mark&compact3` were as fast or faster than those for its copying competitors, it uses less space, and, like all mark-and-sweep collectors, it can accommodate larger sets of accessible data. These space advantages are particularly important for small computers.

Increased space is not free; performance of programs with large memory requirements may suffer because of cache effects and paging. In some environments, programs that use mark-and-sweep collectors have a better locality of reference and hence better cache performance than programs that use copying collectors [16].

The measurements also highlight the importance of collecting strings efficiently and not moving data unnecessarily. `mark&compact3` and `string2` handle strings identically, but `mark&compact3` consistently outperforms `string2` because it does not move data that is already in place. It is especially effective when there is little garbage. Based on these measurements, `mark&compact3` is the best of these alternatives to Icon's current collector.

Garbage collector design continues to depend on many factors, and *a priori* decisions about which collector to use are ill-advised. Collector design is intertwined intimately with the design of other language details from data representations to code generation strategies. Inappropriate collector designs can complicate other parts of a language system unnecessarily and adversely affect performance. For some designs of some languages, copying collectors will indeed provide the best performance. For other designs, however, mark-and-sweep collectors remain viable choices.

Acknowledgements

Chris Fraser provided `burg` and `mkgen` and Ken Walker provided `typsum`, and the referees' perceptive suggestions improved the paper.

References

[1] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[2] Adele Goldberg, David Robson, and Daniel H. H. Ingalls. *SmallTalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[3] David Ungar and Randall B. Smith. SELF: The power of simplicity. *OOPSLA'87 Conference Proceedings, SIGPLAN Notices*, 22(12):227–241, December 1987.

[4] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, February 1989.

[5] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, London, 1992.

[6] Niklaus Wirth. The programming language Oberon. *Software—Practice and Experience*, 18(7):670–690, July 1988.

[7] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[8] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[9] David L. Detlefs. Concurrent garbage collection for C++. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*, chapter 5. MIT Press, 1991.

[10] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[11] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation Techniques*, pages 89–100. MIT Press, Cambridge, MA, 1991.

[12] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[13] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[14] Ralph E. Griswold. Version 8 of Icon. Technical Report 90-1b, Department of Computer Science, The University of Arizona, Tucson, AZ, February 1990.

[15] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, Stanford, CA, February 1988.

[16] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *ACM Conference on LISP and Functional Programming*, pages 87–98, Nice, France, July 1990.

[17] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ, 1986.

[18] Ralph E. Griswold. String scanning in the Icon programming language. *The Computer Journal*, 33(2):98–107, April 1990.

[19] Cary A. Coutant, Ralph E. Griswold, and David R. Hanson. Measuring the performance and behavior of Icon programs. *IEEE Transactions on Software Engineering*, SE-9(1):93–103, January 1983.

[20] Alan Demers, Mark Weiser, Barry Hayes, Hans-Juergen Boehm, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, January 1990.

[21] David R. Hanson. Storage management for an implementation of SNOBOL4. *Software—Practice and Experience*, 7(2):179–192, March 1977.

[22] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[23] Ralph E. Griswold. The Icon program library. Technical Report 90-7b, Department of Computer Science, The University of Arizona, Tucson, AZ, March 1990.

[24] G. David Ripley, Ralph E. Griswold, and David R. Hanson. Performance of storage management in an implementation of SNOBOL4. *IEEE Transactions on Software Engineering*, SE-4(2):130–137, March 1978.

[25] Christopher W. Fraser. A language for writing code generators. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):238–245, July 1989.

[26] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[27] Kenneth Walker. Using the Icon compiler. Icon Project Document IPD157, Department of Computer Science, The University of Arizona, Tucson, AZ, 1991.

[28] David Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19(5):157–167, May 1984.

Appendix

The tables below list the raw data that are displayed in Figures 2–5. Times are in seconds and sizes are in kilobytes.

test program	collector	elapsed time	collection time	number of collections	maximum storage
best	original	21.2	1.15	107	191
	copier	20.1	0.04	215	191
	string	18.3	0.02	143	191
	string2	17.6	0.02	143	191
	mark&compact3	18.9	1.52	71	191
worst	original	12.1	4.41	14	2951
	copier	17.0	8.72	34	5711
	string	18.2	10.22	32	6724
	string2	18.0	10.15	32	6724
	mark&compact3	11.4	4.08	12	2771
string0	original	77.7	0.07	388	191
	copier	76.0	0.08	388	191
	string	73.4	0.06	270	191
	string2	77.4	0.04	271	191
	mark&compact3	77.3	0.05	132	191
string50	original	40.6	25.44	37	3238
	copier	46.0	29.90	47	4516
	string	21.6	6.92	38	6751
	string2	24.5	9.42	38	11919
	mark&compact3	19.9	4.30	15	5026

test program	collector	elapsed time	collection time	number of collections	maximum storage
callgraph	original	3.2	0.31	3	220
	copier	3.4	0.55	7	381
	string	3.0	0.26	7	354
	string2	3.0	0.28	7	379
	mark&compact3	2.8	0.19	2	195
pslist	original	15.3	0.85	9	221
	copier	16.9	2.02	23	376
	string	15.1	0.58	19	396
	string2	14.9	0.64	19	507
	mark&compact3	14.5	0.42	8	257
burg	original	12.1	1.28	9	337
	copier	11.6	1.91	17	542
	string	10.0	0.74	13	458
	string2	10.3	0.79	12	482
	mark&compact3	10.0	0.46	5	254
typsum	original	117.3	35.63	83	2329
	copier	111.1	34.21	100	4592
	string	106.8	26.71	95	4561
	string2	105.3	28.73	94	4599
	mark&compact3	101.7	30.18	78	2304
mkgen	original	49.9	16.22	65	668
	copier	50.9	17.29	84	1302
	string	40.7	6.89	74	1392
	string2	40.2	7.05	65	1816
	mark&compact3	33.1	5.65	44	888
concord	original	59.7	46.88	95	1283
	copier	59.8	47.05	109	2503
	string	20.7	8.21	43	2630
	string2	20.4	8.06	39	3142
	mark&compact3	19.3	6.91	25	1573