Garbage Collection Alternatives for Icon

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

Research Report CS-TR-324-91

May 1991

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.

*Supported by an IBM Graduate Research Fellowship, an AT&T Bell Laboratories Graduate Research Program for Women Grant and a Fannie and John Hertz Foundation Grant.

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 [17], object-oriented languages with late binding times, such as SmallTalk [12] and Self [26], and new languages with traditional compile-time type systems such as ML [2], Eiffel [20], Oberon [28], and Modula-3 [21]. Garbage collectors have also been implemented for languages that were not originally designed to support garbage collection, such as Modula-2 [23], C++ [8], and even C [4].

Recent implementations tend to use *copying* collection algorithms instead of *mark-and-sweep* algorithms [3, 5]. 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 require a second 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 heaps (e.g., tens of megabytes) may amplify the advantages of copying collectors [1].

Recent advances suggest that existing systems with "old" mark-and-sweep collectors might benefit from new collectors. Icon [17] 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 collectors for version 8 of Icon [15].

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. For example, collectors for statically typed languages [9] may not be as well suited to languages with more runtime flexibility. Significant differences can exist even for similar languages. For instance, recent reports on Lisp systems [24, 29] 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. And most systems have complicating idiosyncracies, 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. Source code — if available — can provide many answers, but often these answers are buried in a sea of detail.

Icon's implementation is well documented [16], and the source code is available publicly. The results reported here are, of course, specific to Icon, but Reference [16] 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 [17]. 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, the heap 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

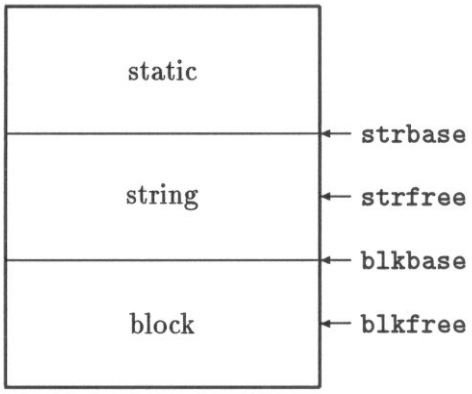

Figure 1: Memory Layout.

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" [14].

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. These blocks have type-specific sizes and layouts and most hold one or more descriptors; Reference [16] gives details. 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 [16]. 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 (including the stack), global variables, and several internal variables. As accessible strings are located, pointers to them 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. During marking, this word 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. Compaction is accomplished by sorting the qualifier list by location and making a pass over the sorted 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 on 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 header words, and, for each such block, traverses this list updating the descriptors with the block's new location. Marked versions of the block codes are also restored in the header words. The second pass compacts accessible blocks, identified by marked block codes, 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. 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 3–76% 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 [6].

The "working set" — the maximum amount of accessible memory used during execution — ranged 100KB to 5MB for the test suite. These sizes are much smaller than the working sets of test suites used in comparative analyses in Lisp, for example, where sizes from 5–100MB are typical. But working sets of 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 working sets may not do so for small working sets.

Long-lived data is data that survives many collections; researchers have long recognized the importance of handling such data efficiently [7, 18, 19]. For the test suite, 30–50% of the accessible data lives to the end of execution. The existing collector does not move data unnecessarily, but a non-generational copying collector will move long-lived 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, 67–97% 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 regions wastes memory for programs that use mostly strings or mostly aggregates. And this division complicates region expansion as described above. For example, executing the 63 programs in the Icon program library [13] generates 9,816 strings with an average length of only 7.23 characters and a median length of 2. Only two percent of these strings were longer than 100 characters; these were counted as 100-character strings. 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. As in all such 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 [3]. 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 eliminates the separate string region and allocates strings and blocks in a single region. It 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 perhaps as much as $N \times M$ bytes. Measuring this expansion helps expose the amount of 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 [18, 22]. 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.

`string2` requires an addition to string descriptors. As above, a string descriptor includes the length of the string and its location. It also includes the offset in words from the head of the string block to the word containing the first character of the string. This offset is used during collection to locate the string block given a descriptor. String operators, e.g., concatenation, update these offsets, but otherwise ignore them. The offset is stored in part of the space previously used for the length, so strings are limited to 65,535 characters.

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

test program	*length in lines*	*execution time in seconds*	*number of collections*	*maximum heap size in KB*
`best`		21	107	191
`worst`		12	14	2951
`string0`		78	388	191
`string50`		41	37	3237
`concord`	53	60	95	1283
`callgraph`	54	3	3	219
`pslist`	426	15	9	221
`mkgen`	991	50	65	668
`typsum`	2804	117	83	2328

Table 1: Test Suite

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 collector [18].

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

Measurement Results

Test Suite

The test suite consists of the nine programs summarized in Table 1, which, for each program, gives its length, and its execution time, number of garbage collections, and heap usage when run with the original collector.

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 collectors (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, 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 causes all entries to be accessible and hence creates *no* garbage,

`string0` and `string50` are similar. `string0` creates 500,000 strings of random lengths between 1 and 100 characters, assigns *none* of them to the entries in `t`, 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 five 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.

`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*.

`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 runtime 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 formatted listings with cross-reference indices. Unlike `callgraph`, which generate 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 [10]. `mkgen` is used to generate the code generators for `lcc` [11]. The input is the VAX specification.

`typsum` reads "ucode," Icon's intermediate representation [16], 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].

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 expanded as necessary during execution.

All times reported are the average elapsed times in seconds on a DECStation 5000, averaged over at least 5 runs. All reported runs achieved at least 90% utilization (i.e., the ratio of times $(user + system)/elapsed \geq 0.90$).[1] In the figures below, all data is normalized so that the original collector runs in unit time. To reduce graphical clutter, the figures display reductions in execution times as percentages, i.e., they display $100 \times (T - X)/T$ where T is the execution time of each test program using the original collector, and X is the execution time using alternative X. The third column in Table 1 gives the values of T.

Space costs are reported as X/S where S is the maximum heap size of each test program using the original collector, and X is the maximum heap size using alternative X. The fifth column in Table 1 gives the values of S.

Figures 2 and 3 show the reductions in execution time for each alternative. Bars extend above the abscissa in proportion to the improvement in execution time; they extend below if execution time increased. The number of collections appears above each bar. 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 heaps are twice as large as `copier`'s. Consequently, they do fewer collections and thus do better than `copier` on `best`. `mark&compact3`'s performance 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 the heap for `best`.

`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

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

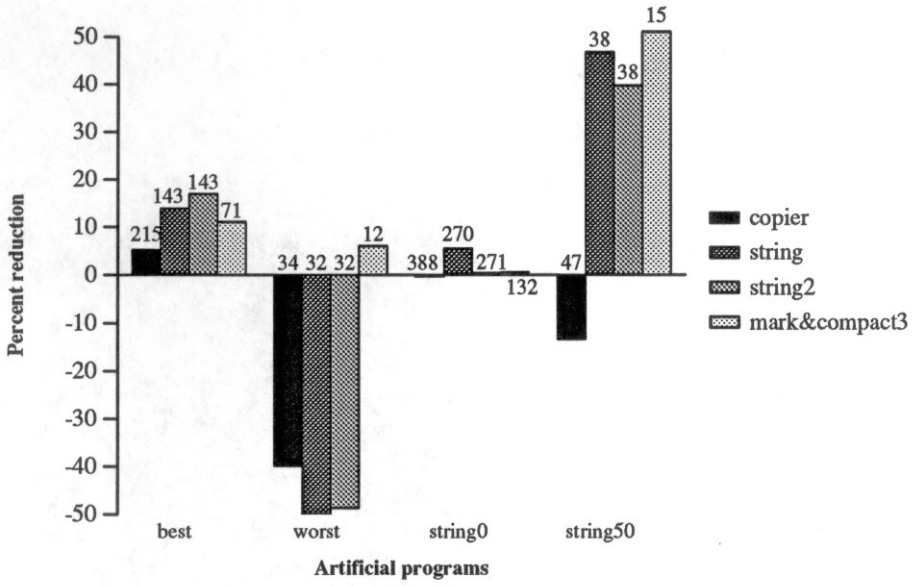

Figure 2: Reduction in Execution Time

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. The performance difference measures the cost of maintaining the offset to the head of the string block in each descriptor, as described above.

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 `typsum`, which does 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 sometimes superior, e.g., on `callgraph`, `mkgen`, and `concord`. `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

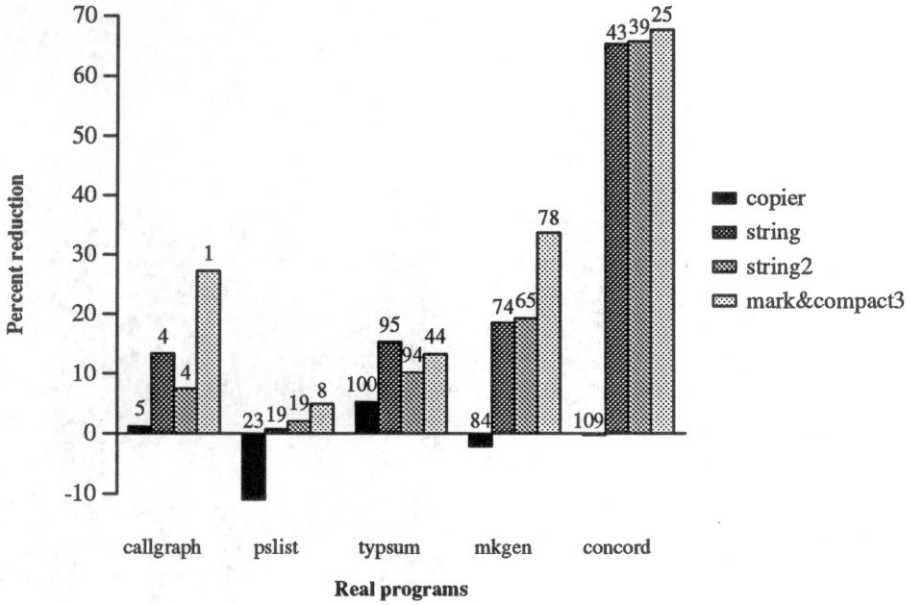

Figure 3: Reduction in Execution Time

described above. Bars extending above the abscissa indicate an increase in memory usage; those that extend below indicate a decrease.

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 all of memory for blocks; the original's string space goes unused for `worst`.

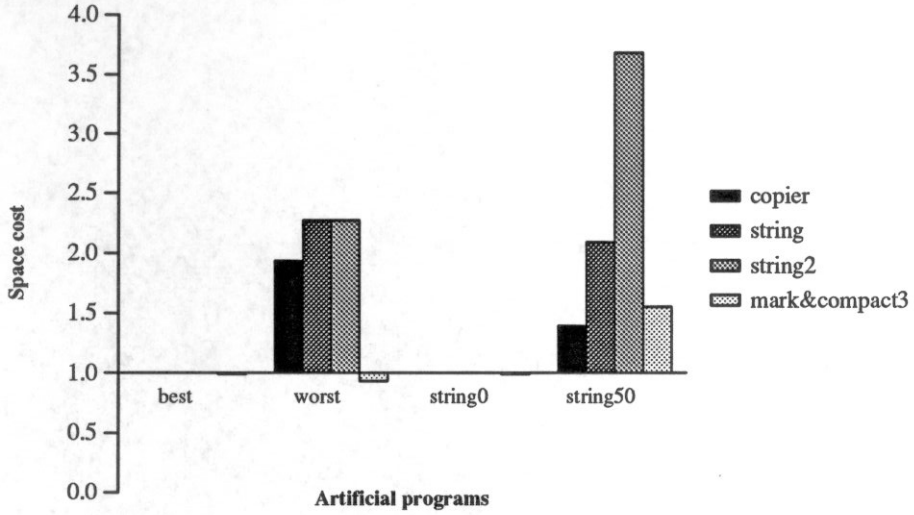

Figure 4: Space Costs

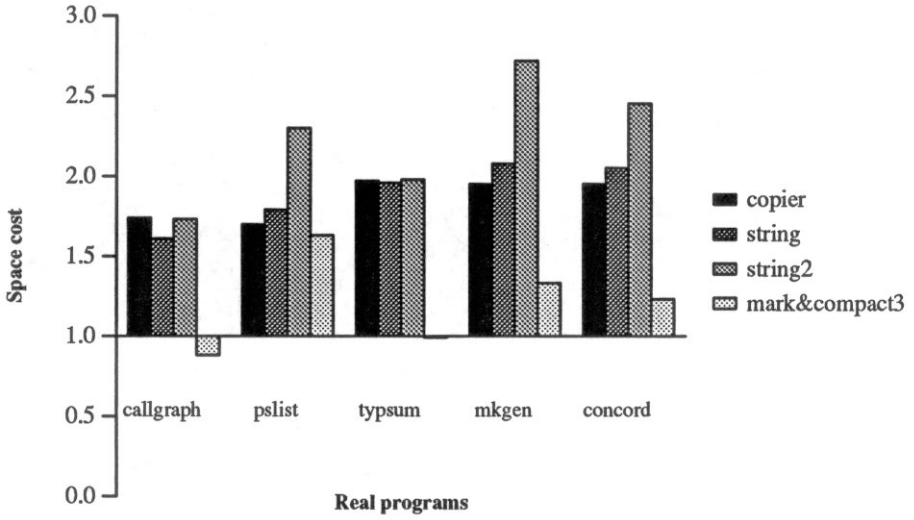

Figure 5: Space Costs

Discussion

As the measurements detailed above demonstrate, a copying collector can improve the execution time of Icon programs that use 1–5MB working sets, but at a significant cost in additional space. For example, `string` and `string2` often yield good execution times, but they have a high space cost. `mark&compact3` yields improvements comparable to or better than its copying competitors, it uses less space, and, like all mark-and-sweep collectors, 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 operating system algorithms. 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 [29].

The measurements also highlight the importance of collecting strings efficiently. Considering both time and space, `mark&compact3` deals with strings better than the copying collectors. Based on these measurements, `mark&compact3` is the best alternative to Icon's current collector.

A generational collector [2, 25] is another alternative that might be explored. A generational collector was not implemented for three reasons. First, generational collectors are most suitable for applicative languages in which assignments are rare [3]. Icon is imperative and assignments abound. Moreover, Icon's goal-directed evaluation mechanism introduces numerous implicit assignments within the runtime system [16].

Second, generational collectors require the cooperation of the compiler to maintain "remembered sets" — lists of old objects that hold pointers to newer objects. Maintaining these sets would require not only changes to Icon's compiler, but extensive changes to its runtime system as well. And since assignments are ubiquitous, these lists are likely to be large.

Finally, measurements of previous collectors with mechanisms similar those used in generational collectors suggest that the potential for improvement is small, at least for languages like SNOBOL4 and Icon [22].

Garbage collector design continues to depend on many factors, and *a priori* decisions to use copying collectors 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, copy-

ing collectors will indeed provide the best performance. For other designs, however, mark-and-sweep collectors may be best.

Acknowledgements

Chris Fraser and Ken Walker provided `mkgen` and `typsum`, respectively.

References

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

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

[3] A. W. Appel. Garbage collection. In P. Lee, editor, *Topics in Advanced Language Implementation Techniques*, chapter 4. MIT Press, 1991.

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

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

[6] C. A. Coutant, R. E. Griswold, and D. R. Hanson. Measuring the performance and behavior of Icon programs. *IEEE Transactions on Software Engineering*, SE-9(1):93–103, Jan. 1983.

[7] A. Demers, M. Weiser, B. Hayes, H.-J. Boehm, D. G. Bobrow, and S. 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, Jan. 1990.

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

[9] J. DeTreville. Experience with concurrent garbage collectors for Modula2+. Research Report 64, System Research Center, Digital Equipment Corp., Palo Alto, CA, Aug. 1990.

[10] C. 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.

[11] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):to appear, Oct. 1991.

[12] A. Goldberg, D. Robson, and D. H. H. Ingalls. *SmallTalk-80: The Language and Its Implementation.* Addison Wesley, Reading, MA, 1983.

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

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

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

[16] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, NJ, 1986.

[17] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[18] D. R. Hanson. Storage management for an implementation of SNOBOL4. *Software—Practice & Experience*, 7(2):179–192, Mar. 1977.

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

[20] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International, London, 1988.

[21] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

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

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

[24] R. A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, CA, Feb. 1988.

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

[26] D. Ungar and R. B. Smith. SELF: The power of simplicity. *OOPSLA '87 Conference Proceedings, SIGPLAN Notices*, 22(12):227–241, Dec. 1987.

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

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

[29] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Symposium on LISP and Functional Programming*, pages 87–98, Nice, France, July 1990.