Empirical Evidence for using Garbage Collection in C and C++ Programs

Dave Detlefs, DEC SRC
July 1993
(Joint work with Al Dosser, DECwest, and Ben Zorn, Colorado)

Position. *C and C++ programmers often believe that garbage collection would impose an unacceptable performance penalty on their programs. Empirical evidence indicates that this belief is often incorrect; there is often no perceivable performance penalty for using garbage collection.*

Overview. Over the past year, I have been part of an advanced development project within DEC aimed at making garbage collection (*GC*) available as part of compilation systems DEC offers for mainstream languages, i.e., C and/or C++. In the best case, this will be seen as a very desirable feature by customers, and help sell a lot of compilers. At the very least, however, we hope to be able to promote the use of GC in internal product development. In order to make this case with intellectual honesty, we tried to gather a number of C and C++ programs that were either Digital products or were used seriously by Digital engineers internally. We built these programs to run with and without garbage collection, and measured their performance on typical workloads. We were particularly interested in *end-to-end* comparisons: whether the use of garbage collection substantially degraded the performance of the programs as seen by users. This paper reports the results of some of these measurements.

I will discuss measurements of three programs: *ild*, an incremental linker derived from the work of Linton's group at Stanford [Quong 91]; *decwrite*, a WYSIWYG document preparation system derived from an early version of *Framemaker;* and *sis*, an interface to a set of digital design tools supported by the University of California at Berkeley [Sentovich 92]. The first two are Digitial products; the last is used extensively by Digital hardware design groups in Massachusetts.

The collector. The collector we're using is derived from the Boehm-Weiser conservative mark-and-sweep collector [Boehm 88], which Hans Boehm has been supporting and improving for several years. The Xerox Corporation has very graciously made this software available without legal encumberence. We have tried to extend and "productize" this collector in several ways: adding the generational and incremental extensions described in [Demers90] and [Boehm 91], making it 64-bit ready, and dealing with the issue of "syscall-safety" when using VM-protection techniques.

ild. The *ild* program has a command-line interface identical to that of *ld*. The difference is that *ild* starts by checking for the existence of a running *ild* server for the executable that the link will create. If none exists, the link proceeds as it would with *ld,* except that at the end of the link, the *ild* process forks, and the child process becomes the server for that target executable, thereby capturing the state of the linking process. The second and subsequent *ild* invocations for that executable find an existing server and pass it their command line, allowing it to decide which object files have changed, and to do the link in an incremental fashion.

The second problem with each of these test cases was developing a repeatable test case. (The first problem, of course, is getting to the point where you can actually build the program you're testing in your environment!) Serendipitously, I was able to fashion a test case for *ild* involving a program I happened to be working on in another context. The environment I was using recorded incremental source changes every time I checked in a version of this program, so I was able to reconstruct the object files corresponding to 13 consecutive versions of my program, and write a script that linked these 13 versions using *ild.*

The results are quite satisfactory to the GC zealot. Here are the results of the Unix *time* command on two pairs of runs, with and without GC. The third number in each line is elapsed time. Try to guess which pair is which:

```
6.554u 14.635s 1:24.39 25.0% 123+510k 507+597io 266pf+0w
6.605u 14.877s 1:32.21 23.2% 123+504k 754+594io 266pf+0w

4.695u 14.655s 1:32.07 21.0% 106+255k 1181+610io 180pf+0w
4.632u 14.159s 1:20.34 23.3% 96+245k 760+591io 153pf+0w
```

The answer is that the first pair of runs is with GC, the second without. The most important point to gather from this test case is that from an end-to-end perspective, the performance of *ild* depends little on the efficiency of storage management -- a linker (even an incremental one) makes heavy use of the file system, so file system and network performance dominate in this test case. The fourth number in these lines is the percentage CPU utilization; this test case is far from CPU-bound. So, it seems safe to say that *ild* performance would not have suffered significantly had it been designed to use GC. (For this audience, it goes without saying that the implementation would have been at least somewhat simplified by using GC.)

DECwrite. *DECwrite,* an X-based interactive program with a graphical user interface, presented some special problems. The first problem was obtaining a repeatable test case. Many extant reports of large-scale measurements of GC performance in interactive programs (see for example, [Ungar 88]) measure long sessions, but not in a manner that allows the same test to be repeated with different collector implementations.

The main problem here is how to capture the input of an interactive program and play it back later to a different version of the program. Fortunately, I found a Digital product that went a significant distance toward solving this problem. Digital's Software Engineering Toolkit includes a tool suite called the DEC Test Manager. One pair of tools is *dxdtmplay* and *dxdtmrecord.* These are based on the *Xtrap* [Jamison 91] extension to X11; *dxdtmrecord* logs X11 input events to a file, and *dxdtmplay* replays such a log, making it seem like a ghostly hand is entering keystrokes and mouse clicks. The log is in an ASCII format, and records pixel coordinates of each mouse event and the time interval in milliseconds between every two events.

I tested this tool on small cases, with fine results. A member of our administrative staff taking a course at a local college was kind enough to let me record a session in which she typed a term paper, resulting in a 1.5 MByte log.

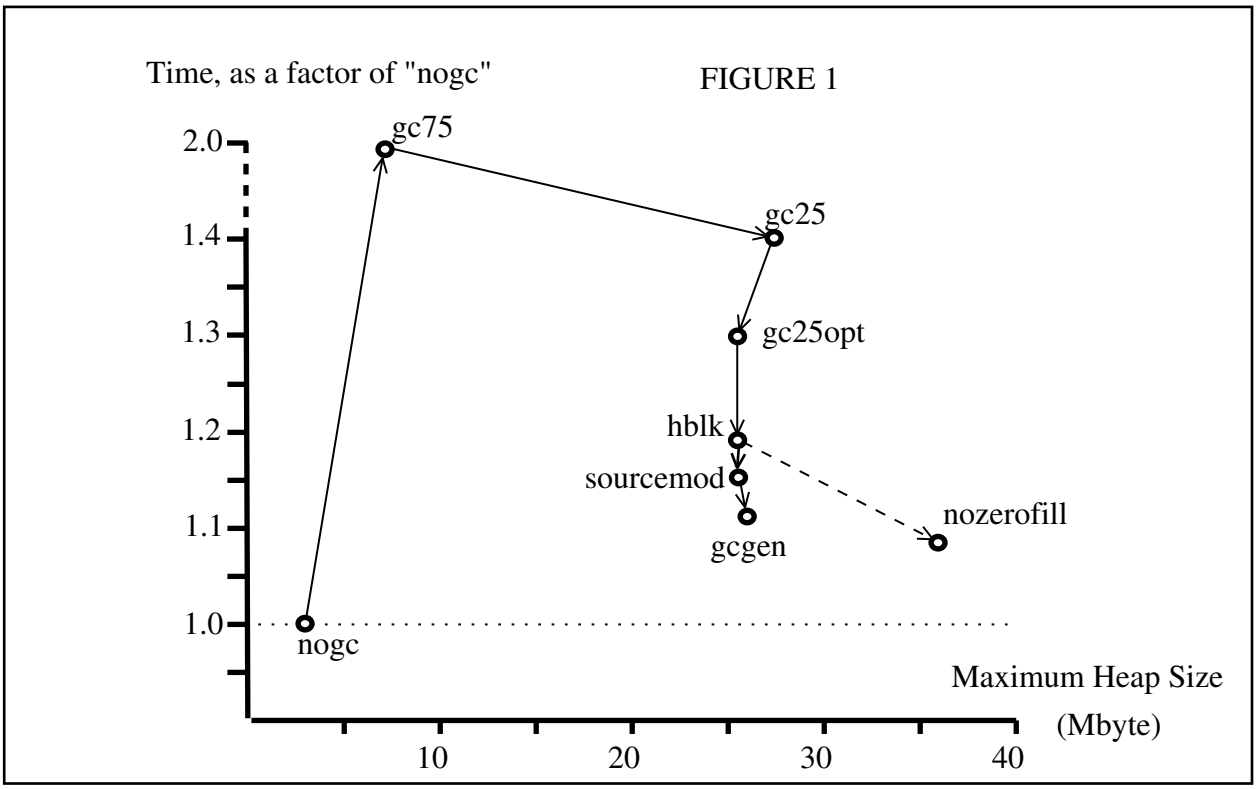
That was the good news; the bad news started when I began to try to replay the log. For playback to work reliably, the log must capture the results of all non-deterministic events in the input. For example, consider a mouse click that pops up a dialog box, followed by another mouse click that selects an option -- the second mouse click must be delayed during playback until the

program has painted the dialog box and is ready for the input, or the X server might deliver the mouse click to the wrong window. The *dxdtm* tools do a good job of recording this kind of output-input synchronization. Some sources of non-determinacy remain, however. *DECwrite* has a scroll bar with up and down arrows; holding the mouse button down on one of these arrows causes smooth scrolling until the mouse button is released. The *dxdtm* log records the duration of the scrolling, but has no way of recording the final position reached in the document. The program will scroll for the same length of time during playback, but may reach a different position, which invalidates the rest of the log.

I was able, through more elbow grease and trial and error than I really want to admit, to translate these non-deterministic time-based actions into deterministic actions based on pixel coordinates. Eventually, I massaged a prefix of the original log into a version that runs for about 1/2 hour and produces a 7-page document, with about a 50% probability of producing the correct result (I have not yet managed to root out all sources of non-determinacy.)

The aim in targetting an interactive program was to provide a benchmark against which to measure the effectiveness of the incremental version of the collector. The bad news is that I can't really claim that we've achieved this goal; the good news is that this doesn't really seem to be necessary. Using the non-incremental collector on this test case, the DECwrite heap grows to about 3 MBytes in the 1/2 hour test, and only 7 collections occur, the longest of which takes less than 1 second (on a DECstation 5000/200 with enough memory to prevent paging.) A user would be hard-pressed to determine whether DECwrite was using collection when working on a document of this size. However, one could very legitimately object that this is not a persuasive case, since editing much larger documents might produce a larger heap, and longer pause times. We think that we can provide acceptable performance for larger heaps as well, but the difficulties in performing this kind of measurement have so far prevented us from gathering any data to support this claim. We will persevere.

sis. In some sense the *sis* program is a worst case example for garbage collection. It is highly cpu-intensive, so there are no spare cycles in which to hide GC. Even on modern hardware, runs of this program can be quite long. We used a realistic input file provided to us by Digital chip designers that took about 1.5 hours to run; in runs of this duration, GC overheads of a few percent translate into significant lengths of absolute time. *sis* makes significant use of the heap: it allocates about 1 byte every 4 instructions. (However, the heap never actually grows very large: the maximum number of heap bytes in use at any time is a little less than 2 Mbyte.) The rest of this section will give a history of our efforts to get good performance on SIS with a conservative mark-and-sweep collector. Figure 1 shows a number of data points, which I will explain.



Point "nogc" represents the original program, without GC. It ran for 92 minutes, using a maximum heap size of 3.5 Mbyte. Point "gc75" represents our initial attempt to run this program using garbage collection. It was rather disappointing (note that the elapsed time scale is **not** contiguous): using garbage collection doubled the elapsed time. The problem was that the default behavior of the collector was to attempt to keep the heap 75% occupied; in the steady state, a collection would leave 75% of the heap occupied and 25% free; the next collection would occur when that free storage had been exhausted. This test performed over 7000 collections!

I decided to "cut us some slack." GC offers a fairly straightforward time-space tradeoff -- using more memory allows less frequent collection, and less overhead. It seems perfectly reasonable to assume that applications like this will be run on memory-rich workstations. This is an input that takes more than an hour to run on a workstation, but requires a heap of only 3.5 Mbytes, a fraction of the memory available in a modern workstation configuration. If the CPU is completely occupied, the memory might as well be used, too. So we switched to a 25% heap occupancy target, resulting in the "gc25" data point. Our time overhead was now down to about 40%, because the number of collections decreased to about 900. Of course, the tradeoff was a significant increase in heap size, but again, if the memory is available, it might as well be used.

The next step is just to show how honest I'm being: I had managed to obfuscate the build procedures to an extent that the first two GC tests were done with an unoptimized version of the collector. Optimizing the collector produced "gc25opt", and time overhead went down to 30%.

The "hblk" point represents the detection and correction of a performance bug in the collector. The Boehm/Weiser allocator (like all good allocators) has two levels: a page allocator and a small-object allocator. Pages are allocated from a free-page list, maintained in ascending address order. The sweep phase of the collection examines heap pages in decreasing address order, putting free pages into the free-page list. Usually, collections occur because the free-page list is empty, so the above procedure always inserts at the head of the free page list. However, the *sis* test case allocated a significant number of large objects, so collections were sometimes necessary when the free-page list was not empty, because it was impossible to find a block of contigous pages large enough to satisfy a request. In this case, the list traversals required to insert free pages into the non-empty sorted free-page list consumed a surprisingly large fraction of the GC overhead. When this was corrected, the overhead decreased to 18%.

The "nozerofill" test case represents an interesting blind alley. The Boehm/Weiser collector zero-fills objects after freeing them. Recall that the collector is completely conservative -- an integer in a register that looks like a pointer value will cause the object to which it "points" to be retained by the collection. If that object had been freed but not zero-filled by the last collection, and was the root of some large linked structure that had also been freed, the entire structure would be retained by the current collection. Zero-filling objects as they are freed prevents this situation. This is purely a performance issue; it doesn't affect correctness. Measurements indicated that significant time was being spent in *bzero,* so I decided to investigate whether zero-filling was a win or not in this case. When I saw the elapsed time number, I was excited; but when I saw the log and the final heap size, my excitement disappeared. Apparently, not performing zero-fill did result in excess storage retention, which grew the heap, resulting in fewer collections (744). We could have obtained similar results simply by further decreasing the occupancy target.

To produce the next step, "sourcemod", I changed the ground rules of the experiments to allow (slight!) modification of the *sis* source code. The collector offers an alternate allocation routine, *GC_malloc_atomic.* By using this routine, the programmer asserts that the allocated object will contain no pointers, and will therefore not be scanned during the marking phase -- it is a leaf node of the pointer graph. Using this routine can obviously save the collector work. With a 15 minute session in *gdb,* I was able to identify a single source line that allocates many objects of a large pointer-free type. I changed that line to use *GC_malloc_atomic.* Another class of source changes that can help is to eliminate code necessary for explicit storage management. To produce "sourcemod", I also commented out the body of a routine that freed all members of some linked list. The result was quite encouraging. A small amount of work shaved off 3 percentage points in the elapsed time.

The final data point is "gcgen"; at this point we had reached a point where we were ready to try out the generational extension to the collector. The performance of this collector is only slightly better than its non-generational counterpart for a similar heap size. It does significantly more collections (1697 in this test), but they are shorter because they mark only from pages modified by the mutator since the last collection.

Summary. It is difficult to do good measurements, and it sometimes seems that every measurement raises more questions than it answers. Still, I think that this work (and Zorn's excellent

related work) represent a good start in demonstrating to C and C++ programmers that garbage collection need not incur a high performance penalty when used in their programs.

Acknowledgements. I'd like to thank Allan Heydon for a detailed reading of this paper that improved it immensely.

References.

[Boehm 88] Boehm, H.-J., and Weiser, M, "Garbage collection in an uncooperative environment", Software Practice and Experience, 18:9, September, 1988, pages 807-820.

[Boehm 91] Boehm, H.-J.; Demers, A. J.; and Shenker, S; "Mostly Parallel Garbage Collection"; in the Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

[Demers 90] Demers, A.; Weiser, M.; Hayes, B.; Boehm, H.; Bobrow, D.; Shenker, S.; "Combining Generational and Conservative Garbage Collection: Framework and Implementations", in the Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages, pages 261-9, January 1990, ACM, New York, NY.

[Jamison 91] Jamison, A.; Miller, K.; Annicchiarico, D.; Chesler, R.; Coutu, D.; Durso, G.; Evans, M.; Henry, M.; Savery, M.; **XTrap**: The XTrap Architecture, Digital Equipment Corporation, July, 1991. (Available as part of the MIT X11 distribution.)

[Sentovich 92] Sentovich, E. M.; Singh, K. J.; Lavango, L.; Moon, C.; Murgai, R.; Saldanha, A.; Savoj, H.; Stephan, P. R.; Brayton, R. K.; and Sangiovanni-Vincentelli, A., "SIS: A System for Sequential Circuit Synthesis", Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California at Berkeley, May, 1992.

[Ungar 88] Ungar, D., and Jackson, F.; "Tenuring Policies for Generation-Based Storage Reclamation", ACM SIGPLAN Notices, 23:11, November, 1988, pages 1-17.

[Quong 91] Quong, R. W., and Linton, M. A.; "Linking Programs Incrementally", ACM Transactions on Programming Languages and Systems, 13:1, January, 1991, pages 1-20.