SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(6), 527–542 (JUNE 1994)

Memory Allocation Costs in Large C and C++ Programs

david detlefs and al dosser
Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301, U.S.A. (e-mail: detlefs@src.dec.com,dosser@zso.dec.com)

and

benjamin zorn
Department of Computer Science, Campus Box #430, University of Colorado, Boulder CO 80309-0430, U.S.A. (e-mail: zorn@cs.colorado.edu)

SUMMARY

Dynamic storage allocation is an important part of a large class of computer programs written in C and C++. High-performance algorithms for dynamic storage allocation have been, and will continue to be, of considerable interest. This paper presents detailed measurements of the cost of dynamic storage allocation in 11 diverse C and C++ programs using five very different dynamic storage allocation implementations, including a conservative garbage collection algorithm. Four of the allocator implementations measured are publicly available on the Internet. A number of the programs used in these measurements are also available on the Internet to facilitate further research in dynamic storage allocation. Finally, the data presented in this paper is an abbreviated version of more extensive statistics that are also publicly available on the Internet.

key words: Garbage collection Dynamic storage allocation Performance evaluation Conservative collection Dynamic memory management

INTRODUCTION

Dynamic storage allocation (DSA) is an important part of many C and C++ programs, including language interpreters, simulators, CAD tools, and interactive applications. Many different algorithms for dynamic storage allocation have been designed, implemented, and compared. In the past, implementation comparisons have most often been based on synthetic allocation behavior patterns, where object lifetime, size, and interarrival time are taken from probability distributions (e.g. References 1 and 2). Recently, Zorn and Grunwald have shown that the use of synthetic behavior patterns may not lead to an accurate estimation of the performance of a particular algorithm.[3]

The existence of instruction-level profiling tools[4,5] has made it possible to directly count the number of instructions required by various allocation algorithms in large, allocation-intensive programs. In this paper, we present a large number of detailed measurements of the performance of five very different dynamic storage allocation

CCC 0038–0644/94/060527–16 Received 23 August 1993
Revised 26 January 1994

implementations in 11 large, allocation-intensive C and C++ programs. The results of this paper extend and complement the C-program measurements of Zorn.[6] The purpose of this paper is to make additional detailed measurements available to a broad audience. For more details about the implementations measured and the measurement methods used, the reader is referred to other papers.

One of the DSA implementations measured is a publicly available conservative garbage collector for C and C++ (BW 2.6+$_{ms,ip}$).[7] Our measurements show that this collector is competitive in both CPU time and memory usage with existing commercial-quality malloc/free allocators. Furthermore, this allocator can be used to replace the operating system calls to malloc/free without any modifications to the program source code and is compatible with both C and C++ programs. We conclude that conservative garbage collection is a competitive alternative to malloc/free implementations, and programmers should keep this technology in mind when building allocation-intensive programs.

PROGRAMS

The programs we measured were drawn from a wide variety of application areas and were written in C and C++. All the programs measured make explicit calls to malloc and free to allocate and deallocate heap storage, respectively. Some of the programs, notably Sis, Espresso, Gawk, and Perl, also make a relatively large number of calls to the realloc function. Table I summarizes the functionality of all of the programs that we measured and Table II summarizes the allocation behavior of those programs. In these and all subsequent tables, the programs are ordered by approximate size in lines of source code.

Many of the programs measured are publicly available, whereas others are proprietary. To allow other researchers to reproduce our results, we have made the tested versions of a number of our test programs available on the Internet. The programs Perl, Ghost, Make, Espresso, Ptc, Gawk, and Cfrac are available via anonymous FTP from the machine ftp.cs.colorado.edu in the directory pub/cs/misc/malloc-benchmarks. A README file in that directory describes how these benchmarks were used to gather the statistics presented. Furthermore, each benchmark includes a number of test inputs, including the ones used in this paper. These programs have been used in a variety of dynamic storage allocation studies (e.g. References 3, 8, and 9).

ALLOCATORS

The allocators we measured are summarized in Table III. Based on our experience, these allocators implement some of the most efficient dynamic storage allocation algorithms currently available. In particular, they are significantly faster than both the Cartesian tree implementation and the first-fit implementation measured by Zorn.[6] Furthermore, four of the five allocators are publicly available and can be obtained via the Internet.

Other recent papers comparing the performance of various aspects of dynamic storage allocation also describe these algorithms, and we refer the interested reader to those papers. Specifically, G++, ***Gnu***′, and ***Qf*** all described in more detail by Grunwald, Zorn and Henderson.[9] The Berkeley Unix 4.2 BSD allocator, of which

Table I. General information about the test programs

Program	Language	Description
Sis	C	SIS, Release 1.1, is a tool for synthesis of synchronous and asynchronous circuits. It includes a number of capabilities such as state minimization and optimization. The input used in the run was the full simplification of an exclusive-lazy-and circuit.
Geodesy	C++	Geodesy, version 1.2, is a programming language debugger. The input to this program is a C++ compiler front end. The operations performed include setting breakpoints, examining the stack, and continuing.
Ild	C++	Ild, version 1.2, is an incremental loader. The input to the program involved incrementally loading 16 saved versions of a set of 26 object files.
Perl	C	Perl 4.10, is a publicly available report extraction and printing language commonly used on UNIX systems. The input script formatted the words in a dictionary into filled paragraphs.
Xfig	C	Xfig, version 2.1.1, is an interactive drawing program. The test case used included the creation of a large number of circles and splines that were duplicated, resized, and reshaped.
Ghost	C	GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input file was a 126-page user manual. This execution of GhostScript did not run as an interactive application as it is often used, but instead was executed with the NODISPLAY option that simply forces the interpretation of the PostScript (without displaying the results).
Make	C	GNU make, version 3.62 is a version of the common ‘make’ utility used on UNIX. The input set was the makefile of another large application.
Espresso	C	Espresso, version 2.3, is a logic optimization program. The input file is an example provided with the release code.
Ptc	C	PTC, version 2.3, is a Pascal to C translator. The input file was a 19,500 line Pascal program (mf2psv.p) that is part of the $\TeX$ release.
Gawk	C	GNU Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script formatted words in a dictionary.
Cfrac	C	CFRAC, version 2.00, is a program to factor large integers using the continued fraction method. The input was a 35-digit product of two large prime numbers.

Ultrix is a derivative, is also described in that paper. The BW 2.6+$_{ms,ip}$ allocator is described in detail by Boehm and Weiser[7] and summarized by Zorn.[6] Black-listing, an enhancement present in Version 2.6 of the collector, is described by Boehm.[11]

The conservative collection allocator (BW 2.6+$_{ms,ip}$) differs significantly from the other allocators in that programmers using it are not required to call free. Instead, the allocator periodically determines what heap objects are no longer in use and automatically frees those objects. As such, this allocator provides significant advantages to users that are not emphasized at all in the performance measurements provided in this paper. Furthermore, because all the application programs measured were written with explicit calls to malloc and free, the BW 2.6+$_{ms,ip}$ allocator in these measurements is placed at somewhat of a disadvantage. Specifically, while the allocator provides a malloc atomic function for allocating objects that do not contain

Table II. Performance information about the memory allocation behavior for each of the test programs. Instructions executed shows the total instructions executed by the program using the *Ultrix* allocator. Total bytes and Total objects refer to the total bytes and objects allocated by each program. Average size shows the average size of the objects allocated. Maximum bytes and Maximum objects show the maximum number of bytes and objects, respectively, that were allocated by each program at any one time

Program	Lines of source	Instructions executed ($\times 10^6$)	Total objects ($\times 10^3$)	Total bytes ($\times 10^3$)	Average size (bytes)	Maximum objects ($\times 10^3$)	Maximum bytes ($\times 10^3$)	Allocation rate (Kbytes/s)
Sis	172000	64794·1	63395	15797173	249·2	48·5	1932·2	4120·8
Geodesy	82500	2648·1	2517	42152	16·7	113·8	3880·6	324·3
Ild	36000	381·3	33	24829	752·4	2·1	1278·7	220·3
Perl	34500	1091·0	1604	34089	21·3	2·3	116·4	714·2
Xfig	30500	52·4	25	1852	72·7	19·8	1129·3	372·1
Ghost	29500	1196·5	924	89782	97·2	26·5	2129·0	1861·7
Make	21000	53·7	23	539	23·0	10·4	208·1	282·5
Espresso	15500	2400·0	1675	107062	63·9	4·4	280·1	1497·1
Ptc	9500	353·9	103	2386	23·2	102·7	2385·8	202·4
Gawk	8500	957·3	1704	67559	39·6	1·6	41·0	2050·1
Cfrac	6000	202·5	522	8001	15·3	1·5	21·4	1145·9

pointers, these programs do not take advantage of that function. Other operations that the test programs perform for the sole purpose of allowing them to correctly call free (e.g., maintaining object reference counts) are also unnecessary when the BW 2.6+$_{\mathit{ms,ip}}$ allocator is used.

RESULTS

The results presented were gathered using a variety of measurement tools on a DECstation 5000/240 with 112 megabytes of memory. Instruction counts were all gathered by instrumenting the programs with Larus' QPT tool,[4,14] which presents per-procedure instruction counts with an output format similar to that of gprof.[15] Program execution time was measured using the Unix C-shell built-in 'time' command. The measurement of each program's live data was gathered using a modified version of malloc/free, and allocator maximum heap sizes were measured using a modified version of the Unix sbrk system call.

In all of the tables presented, we indicate both the absolute performance of each allocator and also the relative performance of each allocator compared to the *Ultrix* allocator. The *Ultrix* allocator was chosen as the baseline for comparison because it is a commercially implemented allocator distributed with a widely used operating system.

The measurements we present concern the CPU overhead (in terms of total execution time and time spent in allocation routines) and memory usage of the various combinations of allocators and programs. Tables IV and V show how many instructions, on average, each program/allocator required to perform the malloc and free operations, respectively. These two tables should be used only for comparing the explicit malloc/free implementations, as substantial overhead in the BW 2.6+$_{\mathit{ms,ip}}$ allocator, resulting from garbage collections, sometimes occurs in the realloc routine,

Table III. General information about the allocators. All the allocators except BW 2.6+$_{ms,ip}$ are described in more detail in Reference 9

Ultrix	*Ultrix* is a variant of the malloc implementation, written by Chris Kingsley, that is supplied with the Berkeley 4.2 Unix release. It is not publicly available, but comes with the DEC Ultrix operating system.
BW 2.6+$_{ms,ip}$	This is version 2.6 of the Boehm–Demers–Weiser conservative garbage collector. With various other authors, Boehm describes a number of related versions of this collector.[7,10,11] For the measurements collected, the definitions of MERGE_SIZES (Ms) and ALL_INTERIOR_POINTERS ($_{Ip}$) were enabled. The most recent version of the collector is version 3.6. Contact Person: Hans Boehm (Hans_Boehm.PARC@xerox.com) FTP Site: anonymous@arisia.xerox.com:/pub/russell/gc.tar.Z
Gnu′	*Gnu*′ is variant hybrid first-fit/segregated algorithm written by Mike Haertel (version dated 930716). It is an ancestor/sibling of the malloc used in GNU libc, but is smaller and faster than the GNU version. Contact person: Mike Haertel (mike@cs.uoregon.edu) FTP Site: anonymous@ftp.cs.uoregon.edu:pub/mike/malloc.tar.z
G++	G++ is an enhancement of the first-fit roving pointer algorithm using bins of different sizes. It is distributed with the GNU C++ library, libg++ (through version 2.4.5) and is also available separately. Contact Person: Doug Lea (dl@oswego.edu) FTP Site: anonymous@g.oswego.edu:/pub/misc/malloc.c
Qf	*Qf* is an implementation of Weinstock and Wulf's fast segregated-storage algorithm based on an array of free lists.[12,13] Like the *Gnu*′ algorithm, *Qf* is a hybrid algorithm that allocates small and large objects in different ways. Large objects are handled by a general algorithm (in this case, G++). Contact Person: Dirk Grunwald (grunwald@cs.colorado.edu) FTP Site: anonymous@ftp.cs.colorado.edu:pub/cs/misc/qf.c

which is not presented. Also note the BW 2.6+$_{ms,ip}$ allocator requires only two instructions per free because we have intentionally caused frees for this allocator to have no effect. In fact, the Boehm–Weiser collector does support explicit programmer frees, but we disabled them to observe the performance of the collection algorithm.

In Table V, we also see that many of the allocators perform a constant number of instructions in free. The *Ultrix* allocator requires 18 instructions to place the freed objects on the appropriate free list. The G++ allocator requires only eight instructions by deferring the size determination of the freed objects until a subsequent malloc. Thus, G++ requires fewer instructions per free than *Ultrix* but more instructions per malloc.

Table VI shows the average number of instructions per object allocated that each program/allocator spent doing storage allocation. This table shows the total instructions in malloc, free, realloc, and any related routines, divided by the total number of objects allocated. This table should be used to compare the per-object overhead of all the allocators, including BW 2.6+$_{ms,ip}$. One should note that the overhead of an allocator has a per-object-allocated and a per-byte-allocated component; for allocators other than BW 2.6+$_{ms/ip}$, the per-object component dominates. If this table showed instructions/allocated-byte, outliers for BW 2.6+$_{ms,ip}$ such as Ild, which allocates a small number of relatively large objects, would be less unusual.

Table IV. Absolute and relative instructions per call to Malloc. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (instr/malloc)		BW 2.6+$_{ms,ip}$ (instr/malloc)		*Gnu*′ (instr/malloc)		G++ (instr/malloc)		*Qf* (instr/malloc)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	60	1·00	489	8·15	101	1·68	84	1·40	192	3·20
Geodesy	46	1·00	179	3·89	81	1·76	53	1·15	29	0·63
Ild	57	1·00	3544	62·18	92	1·61	99	1·74	76	1·33
Perl	46	1·00	207	4·50	82	1·78	44	0·96	37	0·80
Xfig	59	1·00	543	9·20	105	1·78	62	1·05	89	1·51
Ghost	56	1·00	614	10·96	101	1·80	71	1·27	74	1·32
Make	48	1·00	296	6·17	91	1·90	74	1·54	43	0·90
Espresso	50	1·00	208	4·16	90	1·80	77	1·54	40	0·80
Ptc	55	1·00	456	8·29	101	1·84	66	1·20	47	0·85
Gawk	49	1·00	83	1·69	84	1·71	54	1·10	39	0·80
Cfrac	47	1·00	106	2·26	86	1·83	30	0·64	33	0·70
Average	52	1·00	611	11·04	92	1·77	65	1·24	64	1·17

Table V. Absolute and relative instructions per call to Free. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (instr/free)		BW 2.6+$_{ms,ip}$ (instr/free)		*Gnu*′ (instr/free)		G++ (instr/free)		*Qf* (instr/free)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	18	1·00	2	0·11	71	3·94	8	0·44	53	2·94
Geodesy	18	1·00	2	0·11	66	3·67	8	0·44	25	1·39
Ild	18	1·00	2	0·11	74	4·11	8	0·44	42	2·33
Perl	18	1·00	2	0·11	82	4·56	8	0·44	31	1·72
Xfig	18	1·00	2	0·11	71	3·94	8	0·44	39	2·17
Ghost	18	1·00	2	0·11	88	4·89	8	0·44	56	3·11
Make	18	1·00	2	0·11	83	4·61	8	0·44	28	1·56
Espresso	18	1·00	2	0·11	84	4·67	8	0·44	32	1·78
Ptc	18	1·00	2	0·11	113	6·28	8	0·44	90	5·00
Gawk	18	1·00	2	0·11	81	4·50	8	0·44	32	1·78
Cfrac	18	1·00	2	0·11	83	4·61	8	0·44	26	1·44
Average	18	1·00	2	0·11	81	4·53	8	0·44	41	2·29

Table VI. Absolute and relative instructions per object allocated. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (instr/object)		BW 2.6+$_{ms,ip}$ (instr/object)		*Gnu*′ (instr/object)		G++ (instr/object)		*Qf* (instr/object)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	78	1·00	655	8·40	172	2·21	92	1·18	246	3·15
Geodesy	63	1·00	194	3·08	144	2·29	61	0·97	53	0·84
Ild	74	1·00	3806	51·43	161	2·18	107	1·45	115	1·55
Perl	64	1·00	229	3·58	163	2·55	52	0·81	68	1·06
Xfig	63	1·00	671	10·65	119	1·89	64	1·02	97	1·54
Ghost	73	1·00	670	9·18	187	2·56	79	1·08	129	1·77
Make	58	1·00	319	5·50	137	2·36	78	1·34	59	1·02
Espresso	68	1·00	318	4·68	174	2·56	85	1·25	72	1·06
Ptc	55	1·00	485	8·82	101	1·84	66	1·20	47	0·85
Gawk	67	1·00	349	5·21	165	2·46	62	0·93	71	1·06
Cfrac	65	1·00	110	1·69	169	2·60	38	0·58	59	0·91
Average	66	1·00	710	10·20	154	2·32	71	1·07	92	1·35

Each program spends a certain number of instructions outside storage allocation routines doing program-specific work. This number (we will call it the ‘application instructions’) remains constant across all the allocators used. We call the instructions spent doing storage allocation the ‘allocation instructions’. Table VII shows what fraction the allocation instructions are of the application instructions. This percentage provides a good measure for comparing the relative CPU overhead of the different allocators. As is clear from the table, the *Ultrix* and G++ algorithms have the best performance and the BW 2.6+$_{ms,ip}$ collector has the worst performance overall.

Table VIII shows the absolute and relative execution times of the different program/allocator combinations. The times presented are the sums of the user and system times reported by the ‘time’ command. These data were collected from a single run of each program/allocator and thus some variation in execution time, which has not been measured, should be expected. However, our experience collecting similar results indicates that the variation observed between different runs with the same input is not a significant fraction of the total execution time.

In comparing Tables VII and VIII we see some unexpected results, specifically with the Sis program. In particular, the G++ allocator spends more of its instructions executing storage allocation code but executes faster than the *Ultrix* allocator. This behavior is more understandable when we look more closely at the separate components of the execution times as reported by the ‘time’ utility (a breakdown of these times into user and system components is publicly available on the Internet). The user time of Sis *Ultrix* is 3675·7 seconds, whereas the system time is 157·9 seconds. The user time of the Sis G++ allocator is 3688·9 seconds, whereas the system time is just 3·3 seconds. Thus, we see that in user time, the *Ultrix* allocator is the faster allocator just as the data in Table VII would lead us to believe. The added overhead of Sis *Ultrix* is caused by additional time being spent in the operating system. From our measurements we have determined that this overhead is not directly attributable to increased page faults in the *Ultrix* allocator, as one might think. Unfortunately we currently do not have the tools necessary to definitively determine the cause of the added system overhead.

Table IX shows the maximum size of the heap for each program/allocator, as measured by calls to the Unix operating system sbrk system call. To measure this value, an instrumented version of sbrk that maintained a high-water mark was used. As is clear from the table, *Gnu*′, *G*++, and *Qf* are all quite space efficient, whereas *Ultrix* and especially BW 2.6+$_{ms,ip}$ require more space.

Finally, Table X shows the maximum amount of fragmentation that occurred in each program/allocator combination. In this case, fragmentation was measured as the ratio between the maximum heap size (as shown in Table IX) and the maximum bytes that were alive in each program at any time (shown in Table II). Although the average heap expansion of the BW 2.6+$_{ms,ip}$ allocator is almost 2·5 times that of the *Ultrix* allocator, we also note that three programs, namely Perl, Gawk, and Cfrac, contribute significantly to this average. All of these programs require a relatively small heap as indicated in Table II (i.e. 116, 41, and 21 thousand bytes, respectively). Because the BW 2.6+$_{ms,ip}$ allocator allocates space in units of 64 kilobytes, the fragmentation of these programs is somewhat exaggerated. If these programs are not considered in the average, the average heap expansion of the BW 2.6+$_{ms,ip}$ allocator is only 1·53 times that of the *Ultrix* allocator.

Table VII. Percent storage allocation instructions/application instructions. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (% exec. time)		BW 2.6+$_{ms,ip}$ (% exec. time)		*Gnu*$'$ (% exec. time)		G++ (% exec. time)		*Qf* (% exec. time)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	8·2	1·00	69·3	8·45	18·2	2·22	9·8	1·20	26·0	3·17
Geodesy	6·4	1·00	19·6	3·06	14·5	2·27	6·1	0·95	5·3	0·83
Ild	0·6	1·00	37·1	61·83	1·6	2·67	0·9	1·50	0·9	1·50
Perl	9·4	1·00	33·7	3·59	26·1	2·78	7·5	0·80	10·8	1·15
Xfig	3·2	1·00	34·5	10·78	6·6	2·06	3·5	1·09	4·9	1·53
Ghost	6·0	1·00	54·8	9·13	15·3	2·55	6·5	1·08	10·5	1·75
Make	2·6	1·00	14·3	5·50	6·2	2·38	3·5	1·35	2·6	1·00
Espresso	5·0	1·00	23·2	4·64	12·8	2·56	6·2	1·24	5·3	1·06
Ptc	1·6	1·00	14·3	8·94	3·0	1·88	1·9	1·19	1·4	0·87
Gawk	13·6	1·00	65·5	4·82	33·6	2·47	11·9	0·88	14·5	1·07
Cfrac	20·1	1·00	34·1	1·70	52·3	2·60	11·7	0·58	18·3	0·91
Average	6·97	1·00	36·40	11·13	17·29	2·40	6·32	1·08	9·14	1·35

Table VIII. Total program execution time. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (seconds)		BW 2.6+$_{ms,ip}$ (seconds)		*Gnu*′ (seconds)		G++ (seconds)		*Qf* (seconds)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	3833·5	1·00	5867·3	1·53	3932·0	1·03	3692·1	0·96	4581·2	1·20
Geodesy	516·1	1·00	520·8	1·01	518·2	1·00	504·3	0·98	494·1	0·96
Ild	74·2	1·00	75·6	1·02	76·1	1·03	75·3	1·01	75·9	1·02
Perl	47·7	1·00	60·3	1·26	50·1	1·05	49·8	1·04	44·3	0·93
Xfig	7·0	1·00	6·7	0·96	7·1	1·02	7·3	1·05	7·3	1·04
Ghost	48·2	1·00	69·8	1·45	57·0	1·18	48·2	1·00	50·2	1·04
Make	1·9	1·00	2·1	1·11	2·0	1·03	1·8	0·92	2·0	1·07
Espresso	71·5	1·00	82·1	1·15	74·0	1·03	72·6	1·02	72·5	1·01
Ptc	11·8	1·00	13·3	1·13	12·3	1·04	13·2	1·12	11·9	1·01
Gawk	33·0	1·00	51·9	1·57	32·1	0·98	31·2	0·95	27·6	0·84
Cfrac	7·3	1·00	8·3	1·13	8·7	1·19	6·6	0·90	7·0	0·95
Average	423	1·00	614	1·21	434	1·05	409	1·00	489	1·01

Table IX. Maximum heap size. Relative is relative to *Ultrix* = 1

Program	*Ultrix* (Kbytes)		BW 2.6+$_{ms,ip}$ (Kbytes)		*Gnu*′ (Kbytes)		G++ (Kbytes)		*Qf* (Kbytes)	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	3387	1·00	7532	2·22	2363	0·70	2776	0·82	8608	2·54
Geodesy	6487	1·00	7348	1·13	4595	0·71	4928	0·76	4864	0·75
Ild	1800	1·00	3572	1·98	1392	0·77	1520	0·84	1456	0·81
Perl	226	1·00	616	2·73	162	0·72	144	0·64	144	0·64
Xfig	1784	1·00	2436	1·37	1582	0·89	1552	0·87	1576	0·88
Ghost	3541	1·00	5268	1·49	2837	0·80	2632	0·74	2408	0·68
Make	390	1·00	584	1·50	306	0·78	328	0·84	336	0·86
Espresso	792	1·00	1188	1·50	340	0·43	320	0·40	408	0·52
Ptc	3438	1·00	3448	1·00	3414	0·99	3360	0·98	3144	0·91
Gawk	79	1·00	352	4·43	83	1·05	64	0·81	64	0·81
Cfrac	64	1·00	504	7·87	64	1·00	48	0·75	64	1·00
Average	1999	1·00	2986	2·48	1558	0·80	1607	0·77	2097	0·95

Table X. Heap expansion due to fragmentation. Relative is relative to *Ultrix* = 1

Program	*Ultrix* Frag.		BW 2.6+$_{ms,ip}$ Frag.		*Gnu*′ Frag.		G++ Frag.		*Qf* Frag.	
	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Sis	1·79	1·00	3·99	2·22	1·25	0·70	1·47	0·82	4·56	2·54
Geodesy	1·71	1·00	1·94	1·13	1·21	0·71	1·30	0·76	1·28	0·75
Ild	1·44	1·00	2·86	1·98	1·11	0·77	1·22	0·84	1·17	0·81
Perl	1·99	1·00	5·42	2·73	1·42	0·72	1·27	0·64	1·27	0·64
Xfig	1·62	1·00	2·21	1·37	1·43	0·89	1·41	0·87	1·43	0·88
Ghost	1·70	1·00	2·53	1·49	1·36	0·80	1·27	0·74	1·16	0·68
Make	1·92	1·00	2·87	1·50	1·51	0·78	1·61	0·84	1·65	0·86
Espresso	2·90	1·00	4·34	1·50	1·24	0·43	1·17	0·40	1·49	0·52
Ptc	1·48	1·00	1·48	1·00	1·47	0·99	1·44	0·98	1·35	0·91
Gawk	1·98	1·00	8·80	4·43	2·08	1·05	1·60	0·81	1·60	0·81
Cfrac	3·06	1·00	24·09	7·87	3·06	1·00	2·29	0·75	3·06	1·00
Average	1·96	1·00	5·50	2·48	1·56	0·80	1·46	0·77	1·82	0·95

Fragmentation, as measured in Table X, is the result of many different causes. Some of these causes are summarized below:

1. Overhead from the allocator data structures. For example, many allocation algorithms require additional words of storage per object to store information such as the object's status (allocated or free) and the object's size.
2. Internal fragmentation. This fragmentation results from rounding up allocation request sizes to common sizes (e.g., some algorithms round requests up to sizes that are powers of two).
3. External fragmentation. This fragmentation results from the available free space being split into pieces that are too small to satisfy most requests, thus making the space unusable. Clearly if request sizes are rounded up, then the result is more internal fragmentation and less external fragmentation.

In addition to the forms of fragmentation that are common to all allocation algorithms, the BW 2.6+$_{ms,ip}$ allocator incurs additional sources of fragmentation summarized below:

1. Storage required to prevent frequent garbage collections. The CPU overhead of the BW 2.6+$_{ms,ip}$ allocator is reduced if garbage collections occur less frequently. However, since no storage is reclaimed between collections, less frequent collections result in a larger heap. Another way to view this form of fragmentation is to observe that the collector only reclaims inaccessible storage when the algorithm runs, and as such unreachable objects are not available for reuse until the collector is invoked.
2. Additional storage that is retained due to collector conservatism. The BW 2.6+$_{ms,ip}$ allocator will preserve any object in the heap that 'appears' to have a pointer pointing to it. Sometimes integer variables contain values that appear to be pointers and the collector conservatively preserves such objects.
3. Additional storage allocated due to the heap-expansion granularity. As mentioned above, the BW 2.6+$_{ms,ip}$ allocator grows the heap in units of 64 kilobytes, even when all the storage allocated may not be used. The other allocators typically have a smaller unit of expansion (e.g., 8 kilobytes).
4. Black-listed blocks in the heap. The BW 2.6+$_{ms,ip}$ allocator uses a heuristic called 'black-listing' to reduce the amount of incorrectly retained garbage. As a result, some of the blocks in the heap are deemed inappropriate for heap-allocation and contribute to overall storage usage.
5. Storage that is 'freed' but remains accessible. In some cases, a programmer will correctly free an object that still has pointers pointing to it. In such cases, the conservative collector (as used in this paper) will ignore the free and reclaim the object only after the last pointer to the object has been removed.

Unfortunately, we were not able to identify the specific sources of fragmentation in the programs and allocators measured. In the future, we may instrument the allocators to collect more specific information about the sources of fragmentation.

SUMMARY

This paper presents detailed measurements of the cost of dynamic storage allocation (DSA) in 11 diverse C and C++ programs using five very different dynamic storage allocation implementations, including a conservative garbage collection algorithm. The measurements include the CPU overhead of storage allocation and the memory usage of the different allocators. All of the DSA implementations we measured are highly efficient, well-crafted programs. Furthermore, four out of five of these implementations are publicly available on the Internet and we provide Internet sites and the contact persons who are responsible for them. Likewise, seven of the eleven programs measured are also available on the Internet, and we provide their location as well. It is our hope that when other researchers implement new algorithms, they will use the programs, allocators, and techniques used in this paper to provide comparable measurements of the new algorithm. We see these programs and allocators not as the final word in allocator benchmarking, but as a first small step along the way.

The data presented in this paper are a subset of data available in a textual form on the Internet. These data are available via anonymous FTP from the machine ftp.cs.colorado.edu in the file pub/cs/misc/malloc-benchmarks/SPE-MEASUREMENTS.txt. Further results will be added to this file as they become available. Please feel free to use these data but we would appreciate your sending one of us e-mail (zorn@cs.colorado.edu) indicating that you intend to use the data and how you intend to use it.

acknowledgements

We would like to thank Hans Boehm, Michael Haertel, and Doug Lea for all implementing very efficient dynamic storage allocation algorithms and making them available to the public. We also thank them for their comments on drafts of this paper. We would also like to thank the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9121269 and by Digital Equipment Corporation External Research Grant No. 1580.

REFERENCES

1. Donald E. Knuth, *Fundamental Algorithms, volume 1 of The Art of Computer Programming,* Addison Wesley, Reading, MA, 2nd edn, 1973, chapter 2, pp. 435–451.
2. David G. Korn and Kiem-Phong Vo, ‘In search of a better malloc’, *Proceedings of the Summer 1985 USENIX Conference*, 1985, pp. 489–506.
3. Benjamin Zorn and Dirk Grunwald, ‘Evaluating models of memory allocation’, *ACM Trans. Modeling and Computer Simulation,* **4**, (1), (1994).
4. Thomas Ball and James R. Larus, ‘Optimally profiling and tracing programs’, *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, January 1992, pp. 59–70.
5. Digital Equipment Corporation, *Unix Manual Page for pixie,* ULTRIX V4.2 (rev 96) edition, September 1991.
6. Benjamin Zorn. ‘The measured cost of conservative garbage collection’, *Software—Practice and Experience,* **23**, 733–756 (1993).
7. Hans-Juergen Boehm and Mark Weiser, ‘Garbage collection in an uncooperative environment’, *Software—Practice and Experience,* **18**, 807–820 (1988).
8. Dirk Grunwald and Benjamin Zorn, ‘CustoMalloc: efficient synthesized memory allocators’, *Software—Practice and Experience,* **23**, 851–869 (1993).
9. Dirk Grunwald, Benjamin Zorn and Robert Henderson, ‘Improving the cache locality of memory allocation, *SIGPLAN’93 Conference on Programming Language Design and Implementation,* Albuquerque, June 1993, pp. 177–186.

10Hans-Juergen Boehm, Alan Demers and Scott Shenker, ‘Mostly parallel garbage collection’, *Proceedings of the SIGPLAN’92 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 157–164.

REFERENCES

11. Han-Juergen Boehm, ‘Space efficient conservative garbage collection’, *SIGPLAN’93 Conference on Programming Language Design and Implementation*, Albuquerque, June 1993, pp. 197–206.
12. Thomas Standish, *Data Structures Techniques*, Addison-Wesley Publishing Company, 1980.
13. Charles B. Weinstock and William A. Wulf, ‘Quickfit: an efficient algorithm for heap storage allocation’, *ACM SIGPLAN Notices*, **23**, (10), 141–144 (1988).
14. James R. Larus and Thomas Ball, ‘Rewriting executable files to measure program behavior’, *Technical Report 1083*, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, March 1992.
15. Susan L. Graham, Peter B. Kessler and Marshall K. McKusick, ‘An execution profiler for modular programs’, *Software—Practice and Experience,* **13**, 671–685 (1983).