Empirical Measurements of Six Allocation-Intensive C Programs; CU-CS-604-92

Benjamin G. Zorn
University of Colorado Boulder

Dirk C. Grunwald
University of Colorado Boulder

Follow this and additional works at: https://scholar.colorado.edu/csci_techreports

Recommended Citation
https://scholar.colorado.edu/csci_techreports/579

This Technical Report is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Empirical Measurements of Six Allocation-intensive C Programs

Benjamin Zorn    Dirk Grunwald
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-604-92     July 1992

University of Colorado at Boulder

Technical Report CU-CS-604-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309
Empirical Measurements of Six 
Allocation-intensive C Programs* 

Benjamin Zorn  Dirk Grunwald 
Department of Computer Science 
Campus Box #430 
University of Colorado, Boulder 80309-0430 

July 1992 

Abstract 
Dynamic memory management is an important part of a large class of computer programs and high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. This paper presents empirical data from a collection of six allocation-intensive C programs. Extensive statistics about the allocation behavior of the programs measured, including the distributions of object sizes, lifetimes, and interarrival times, are presented. This data is valuable for the following reasons: first, the data from these programs can be used to design high-performance algorithms for dynamic memory management. Second, these programs can be used as a benchmark test suite for evaluating and comparing the performance of different dynamic memory management algorithms. Finally, the data presented gives readers greater insight into the storage allocation patterns of a broad range of programs. The data presented in this paper is an abbreviated version of more extensive statistics that are publically available on the internet.

1 Introduction

This paper presents empirical data about the allocation behavior of six allocation-intensive C programs. The data presented describes the distributions of objects sizes, holding times, and interarrival times in each of the programs measured. This data is valuable to designers of dynamic memory management (DMM) algorithms for the following reasons:

- It has been long observed that tailoring a DMM algorithm to the observed empirical behavior of programs results in a more efficient algorithm [9]. The data from these programs provides designers with specific information about the allocation behavior of a broad class of C/Unix programs.

- Such data, specifically about the allocation behavior of C programs, is rarely published.

- The programs measured in this study provide DMM algorithm designers with a test suite of allocation-intensive programs with which they can fairly compare alternative DMM algorithms. We have already used these programs for this purpose [6].

- Such data provides readers with insight into the allocation behavior of a broad class of programs. This insight may help the reader understand the trade-offs in DMM implementations, including the relative advantages and disadvantages of using automatic memory management techniques such as garbage collection.

*This material is based upon work supported by the National Science Foundation under Grants No. CCR-9010624, CCR-9121269, and CDA-8922510
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CFRAC</strong></td>
<td>Cfrac is a program to factor large integers using the continued fraction method. The input is a 22-digit number that is the product of two primes.</td>
</tr>
<tr>
<td><strong>ESPRESSO</strong></td>
<td>Espresso, version 2.3, is a logic optimization program. The input file was one of the larger example inputs provided with the release code (cps).</td>
</tr>
<tr>
<td><strong>GHOSTSCRIPT</strong></td>
<td>GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input used is the Users Guide to the GNU C++ Libraries (126 pages). 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.</td>
</tr>
<tr>
<td><strong>GAWK</strong></td>
<td>Gnu Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script processes a large file containing numeric data, computing statistics from that file.</td>
</tr>
<tr>
<td><strong>PERL</strong></td>
<td>Perl 4.10, is a publicly available report extraction and printing language, commonly used on UNIX systems. The input used was a perl script that reorganizes internet domain names located in the file /etc/hosts.</td>
</tr>
<tr>
<td><strong>CHAM</strong></td>
<td>Chameleon is an N-level channel router for multi-level printed circuit boards. The input file was one of the example inputs provided with the release code (exf). We also measured another channel router (YACR), but the results obtained were not significantly different that those from CHAM.</td>
</tr>
</tbody>
</table>

Table 1: General Information about the Test Programs

Dynamic memory management has always been an important part of a large class of computer programs. Recently, interest in this field has increased, as evidenced by the number of workshops devoted entirely to the subject (garbage collection workshops at recent Object-oriented Programming Languages and Systems (OOPSLA) Conferences and the 1992 International Workshop on Memory Management to name a few). One reason for this increased interest is that object-oriented design encourages programming with large interconnected dynamic structures and broadens the class of programs that use dynamic memory allocation. The increasing use of dynamic memory management brings with it the need to reevaluate the performance of old algorithms for memory management and consider new ones.

Many studies of the relative performance of DMM algorithms have been published through the years. In addition to comparing the performance of DMM algorithms, a number of these papers also present empirical measurements of the allocation behavior of particular programs or systems. In particular, in 1971 Margolin et al presented empirical measurements of the dynamic allocation patterns they observed in a time-sharing operating system [9]. This empirical data has been used in subsequent DMM measurement studies, some as recent as 1985 [10]. Batson et al present empirical data concerning the distribution of program segment sizes in the B5500 operating system [2]. In his book Data Structure Techniques, Standish [11] presents data from Charles Weinstock’s thesis [12] showing the distribution of size requests in the BLISS/11 compiler. Standish also mentions that Weinstock uses Batson’s empirical data in comparing the performance of different DMM algorithms. In a later paper, Batson and Brundage present empirical data about segment sizes and holding times collected from Algol-60 programs [1].

More recently, in 1984 Bozma et al [3] compared the performance of a large number of DMM algorithms based on empirical data gathered from several days of execution on several different multiuser operating systems. In the paper, they present the empirical data gathered, including the average interarrival time and holding time for each of the block sizes allocated. This data represents some of the most complete information published to date, and has been used in a 1989 performance evaluation of DMM algorithms [4]. Most recently, DeTreville has published the results of extensive empirical measurements of heap usage in the Topaz computing environment [5].

From this discussion, we can conclude two things. First, empirical measurements of actual programs are valuable in both designing and evaluating DMM algorithms. For as long as these algorithms have been proposed and evaluated, empirical data has been used in the measurement process. Second, there is a relative lack of empirical data, as evinced by the use of the Margolin data 14 years after it was...
<table>
<thead>
<tr>
<th>Program</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>6,000</td>
<td>15,500</td>
<td>29,500</td>
<td>8,500</td>
<td>34,500</td>
<td>7,500</td>
</tr>
<tr>
<td>Execution Time (instructions ×10⁶)</td>
<td>66.9</td>
<td>61.1</td>
<td>159.2</td>
<td>17.6</td>
<td>33.5</td>
<td>87.1</td>
</tr>
<tr>
<td>Objects Allocated</td>
<td>227,091</td>
<td>186,636</td>
<td>108,550</td>
<td>32,165</td>
<td>26,390</td>
<td>103,548</td>
</tr>
<tr>
<td>Max Objects Allocated</td>
<td>1,231</td>
<td>2,959</td>
<td>6,195</td>
<td>2,447</td>
<td>483</td>
<td>103,413</td>
</tr>
<tr>
<td>Bytes Allocated</td>
<td>3,339,166</td>
<td>14,641,338</td>
<td>18,767,795</td>
<td>722,970</td>
<td>790,801</td>
<td>2,927,254</td>
</tr>
<tr>
<td>Max Bytes Allocated</td>
<td>17,395</td>
<td>136,966</td>
<td>467,739</td>
<td>63,834</td>
<td>24,452</td>
<td>2,711,158</td>
</tr>
</tbody>
</table>

| Size Classes (SC)       | 22    | 328     | 177         | 48   | 79   | 22   |
| Mean Size               | 14.7  | 78.4    | 172.9       | 22.5 | 30.0 | 28.2 |
| Median Size             | 14    | 28      | 116         | 24   | 32   | 24   |
| Mode Size               | 14    | 24      | 116         | 24   | 32   | 24   |

| Interarrival Time Classes (ITC) | 911 | 13,425 | 3,502 | 856 | 587 | 4,316 |
| Mean Interarrival Time      | 295 | 3,273  | 1,465 | 548 | 1,271 | 825 |
| Median Interarrival Time    | 241 | 96     | 1,121 | 451 | 207 | 482 |
| Mode Interarrival Time      | 37  | 15     | 69    | 122 | 44  | 66   |

| Holding Time Classes (HTC) | 12,748 | 76,299 | 15,339 | 5,638 | 5,053 | 13,169 |
| Mean Holding Time          | 172,000 | 467,000 | 5,670,000 | 1,287,000 | 574,000 | 565,000 |
| Median Holding Time        | 600    | 25,593 | 794    | 89   | 6,137 | 479   |
| Mode Holding Time          | 600    | 37     | 236    | 86   | 1570  | 99    |

Table 2: Test Program Performance Information. The SC, ITC, and HTC values indicate the number of distinct size, interarrival times, and holding times respectively in each of the sample programs. All times are presented in instructions.

published. In general, because empirical measurements of dynamic storage allocation in systems are relatively scarce, algorithm evaluators must and do take whatever is available. The intent of this paper is to make additional empirical measurements of allocation-intensive programs widely available. In a companion paper we investigate how different models of allocation behavior, based on empirical data as presented here, can be used to accurately evaluate DMM algorithms [14].

The remainder of this paper has the following organization: Section 2 describes the programs we have measured and Section 3 presents empirical measures of the programs, including distribution of object sizes, object holding times, and object interarrival times. In Section 4, we conclude by summarizing our data and indicating how it can be obtained on the internet.

2 Programs

To gather the data for this paper, we instrumented six allocation-intensive C programs, described in Table 1. The programs represent a wide range of memory intensive tasks, including number factoring, interpreters, logic optimizers, and CAD/VLSI tools. In each case, we actually collected data from at least two input data sets for each program. In order to simplify the presentation here, we show data from only one of these input data sets. Data from all input sets is publically available via the internet.
The data was gathered by tracing the execution of each program using AE [8] on a Sun SPARC processor. AE is an efficient program tracing tool that captures all instruction and data references, as well as special indicators for calls to the malloc and free procedures used for memory allocation. These large, complex traces were distilled to a time-ordered memory allocation trace including only calls to malloc and free. Each memory allocation trace event was time-stamped using the number of instructions since the beginning of the program.

The version of CHAM that we measured does not release much of its allocated memory by calling free. For this program, we monitored the data references of the traced program, and artificially deallocated memory when it was no longer referenced. The artificial free events were inserted in the memory allocation trace, essentially modeling perfect memory deallocation.

After gathering the “allocation trace” of allocate and free events, we measured three data distributions from each trace: object size, object holding time, and object interarrival time. These three distributions capture the allocation behavior of the programs, allowing their empirical behavior to be characterized.

Table 2 summarizes the vital statistics of each program and presents some basic measures of each of these distributions, including the total number of objects and bytes allocated by each program, as well as the maximum number of objects and bytes allocated at any one time by each program. The programs are allocation-intensive, allocating an object every 300–3000 instructions. The fraction of time spent doing memory allocation for these programs depends on the DMM implementation used, but ranges from approximately 5–30%.

The table shows the mean, median, and mode for each of the three distributions mentioned. In most cases, the median is much smaller than the mean, indicating that the distribution is greatly skewed by a very large range of values. As the table shows, the size distribution is less skewed than the others, while the holding time distribution is greatly skewed. Even the size distribution is heavily skewed to smaller objects, the table indicating that in all but one program the most common size class is 32-bytes or smaller.

The table also shows the number of distinct sizes, interarrival times, and holding times (SC, ITC, HTC, respectively). These values indicate that the number of distinct values observed in each distribution is small compared to the total number of instructions executed by each program. For the interarrival times and holding time distribution, this result suggests that program behavior is relatively regular, resulting in a smaller number of distinct classes.

## 3 Data

This section presents more complete information about the observed data distributions in the test programs. In previous work, the distribution of object sizes has been the most widely reported distribution, partly because it is the easiest to measure of the three. Here, we describe the distributions of object size, interarrival time (IAT), and holding time (HT) equally.

The first set of tables (Tables 3, 4, and 5) show how many distinct classes of size, IAT, and HT are required to cover from 50% to 100% of the observed data. This coverage data indicates how skewed the distributions are and how many classes really represent the important aspects of program behavior.

The tables show that most of the data in each distribution is accounted for by a small number of classes. In particular, Table 3 shows that at most two size classes are required to cover 50% of the observed data in all of the test programs. Furthermore, 95% of the observed data is covered by at most 34 classes. Algorithms such as Oldsoo’s adaptive exact-fit allocator [10] and our own CUSTOMALLOC [6] exploit this empirical behavior by adapting allocation policies to the most commonly observed object sizes.

Interestingly, we observe that a small number IAT classes and HT classes also account for a large percentage of the allocation in many of the programs measured. This empirical result, which indicates a large degree of regularity, has yet to be exploited by proposed DMM algorithms.

One important statistical characterization of a distribution is a listing of the quantiles. We present the 5% quantiles of the three distributions in Tables 6, 7, and 8.
<table>
<thead>
<tr>
<th>Program</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>75%</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>90%</td>
<td>5</td>
<td>14</td>
<td>7</td>
<td>2</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>95%</td>
<td>6</td>
<td>34</td>
<td>11</td>
<td>4</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>99%</td>
<td>6</td>
<td>88</td>
<td>19</td>
<td>13</td>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>100%</td>
<td>22</td>
<td>328</td>
<td>177</td>
<td>48</td>
<td>79</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 3: Size Classes Required to Cover Percentages of All Objects in the Test Programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>7</td>
<td>12</td>
<td>31</td>
<td>4</td>
<td>6</td>
<td>44</td>
</tr>
<tr>
<td>75%</td>
<td>25</td>
<td>222</td>
<td>119</td>
<td>34</td>
<td>15</td>
<td>320</td>
</tr>
<tr>
<td>90%</td>
<td>53</td>
<td>1955</td>
<td>413</td>
<td>140</td>
<td>43</td>
<td>1018</td>
</tr>
<tr>
<td>95%</td>
<td>85</td>
<td>5002</td>
<td>849</td>
<td>241</td>
<td>79</td>
<td>1573</td>
</tr>
<tr>
<td>99%</td>
<td>185</td>
<td>11560</td>
<td>2418</td>
<td>547</td>
<td>325</td>
<td>3282</td>
</tr>
<tr>
<td>100%</td>
<td>911</td>
<td>13425</td>
<td>3502</td>
<td>856</td>
<td>587</td>
<td>4316</td>
</tr>
</tbody>
</table>

Table 4: IAT Classes Required to Cover Percentages of All Objects in the Test Programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>24</td>
<td>9730</td>
<td>13</td>
<td>2</td>
<td>41</td>
<td>93</td>
</tr>
<tr>
<td>75%</td>
<td>187</td>
<td>29640</td>
<td>194</td>
<td>115</td>
<td>838</td>
<td>641</td>
</tr>
<tr>
<td>90%</td>
<td>2823</td>
<td>57636</td>
<td>4484</td>
<td>2422</td>
<td>2416</td>
<td>3993</td>
</tr>
<tr>
<td>95%</td>
<td>5230</td>
<td>66968</td>
<td>9912</td>
<td>4030</td>
<td>3734</td>
<td>7992</td>
</tr>
<tr>
<td>99%</td>
<td>10478</td>
<td>74433</td>
<td>14254</td>
<td>5317</td>
<td>4790</td>
<td>12134</td>
</tr>
<tr>
<td>100%</td>
<td>12748</td>
<td>76299</td>
<td>15339</td>
<td>5638</td>
<td>5053</td>
<td>13169</td>
</tr>
</tbody>
</table>

Table 5: HT Classes Required to Cover Percentages of All Objects in the Test Programs.
<table>
<thead>
<tr>
<th>Quantile</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (min)</td>
<td>4</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>24</td>
<td>29</td>
<td>4</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>24</td>
<td>36</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>24</td>
<td>40</td>
<td>24</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>24</td>
<td>64</td>
<td>24</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>25</td>
<td>12</td>
<td>24</td>
<td>116</td>
<td>24</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
<td>24</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>35</td>
<td>14</td>
<td>24</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>40</td>
<td>14</td>
<td>28</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>45</td>
<td>14</td>
<td>28</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>50 (median)</td>
<td>14</td>
<td>28</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>55</td>
<td>14</td>
<td>28</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>60</td>
<td>14</td>
<td>28</td>
<td>116</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>65</td>
<td>16</td>
<td>28</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>70</td>
<td>16</td>
<td>32</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>75</td>
<td>18</td>
<td>32</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>80</td>
<td>18</td>
<td>32</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>85</td>
<td>20</td>
<td>40</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>90</td>
<td>20</td>
<td>64</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>95</td>
<td>20</td>
<td>200</td>
<td>260</td>
<td>24</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>100 (max)</td>
<td>266</td>
<td>19680</td>
<td>20016</td>
<td>8192</td>
<td>5632</td>
<td>36788</td>
</tr>
</tbody>
</table>

Table 6: Quantiles of Object Sizes in the Test Programs. All sizes are in bytes.

<table>
<thead>
<tr>
<th>Quantile</th>
<th>CFRAC</th>
<th>ESPRESSO</th>
<th>GHOSTSCRIPT</th>
<th>GAWK</th>
<th>PERL</th>
<th>CHAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (min)</td>
<td>8</td>
<td>5</td>
<td>67</td>
<td>9</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>37</td>
<td>15</td>
<td>69</td>
<td>14</td>
<td>44</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>37</td>
<td>15</td>
<td>113</td>
<td>73</td>
<td>44</td>
<td>66</td>
</tr>
<tr>
<td>15</td>
<td>37</td>
<td>15</td>
<td>475</td>
<td>122</td>
<td>44</td>
<td>66</td>
</tr>
<tr>
<td>20</td>
<td>46</td>
<td>41</td>
<td>737</td>
<td>122</td>
<td>89</td>
<td>66</td>
</tr>
<tr>
<td>25</td>
<td>46</td>
<td>47</td>
<td>812</td>
<td>122</td>
<td>99</td>
<td>186</td>
</tr>
<tr>
<td>30</td>
<td>118</td>
<td>50</td>
<td>932</td>
<td>122</td>
<td>99</td>
<td>250</td>
</tr>
<tr>
<td>35</td>
<td>152</td>
<td>61</td>
<td>958</td>
<td>224</td>
<td>127</td>
<td>343</td>
</tr>
<tr>
<td>40</td>
<td>184</td>
<td>65</td>
<td>994</td>
<td>299</td>
<td>127</td>
<td>416</td>
</tr>
<tr>
<td>45</td>
<td>228</td>
<td>87</td>
<td>1051</td>
<td>446</td>
<td>155</td>
<td>446</td>
</tr>
<tr>
<td>50 (median)</td>
<td>241</td>
<td>96</td>
<td>1121</td>
<td>451</td>
<td>207</td>
<td>482</td>
</tr>
<tr>
<td>55</td>
<td>252</td>
<td>128</td>
<td>1295</td>
<td>451</td>
<td>207</td>
<td>516</td>
</tr>
<tr>
<td>60</td>
<td>271</td>
<td>226</td>
<td>1639</td>
<td>451</td>
<td>279</td>
<td>577</td>
</tr>
<tr>
<td>65</td>
<td>293</td>
<td>290</td>
<td>1773</td>
<td>475</td>
<td>650</td>
<td>633</td>
</tr>
<tr>
<td>70</td>
<td>306</td>
<td>332</td>
<td>1821</td>
<td>562</td>
<td>912</td>
<td>722</td>
</tr>
<tr>
<td>75</td>
<td>350</td>
<td>475</td>
<td>1980</td>
<td>764</td>
<td>995</td>
<td>853</td>
</tr>
<tr>
<td>80</td>
<td>395</td>
<td>776</td>
<td>2138</td>
<td>1046</td>
<td>2403</td>
<td>1000</td>
</tr>
<tr>
<td>85</td>
<td>560</td>
<td>2062</td>
<td>2310</td>
<td>1046</td>
<td>2491</td>
<td>1119</td>
</tr>
<tr>
<td>90</td>
<td>654</td>
<td>5510</td>
<td>2578</td>
<td>1175</td>
<td>4050</td>
<td>1444</td>
</tr>
<tr>
<td>95</td>
<td>803</td>
<td>8852</td>
<td>3068</td>
<td>1608</td>
<td>5206</td>
<td>1919</td>
</tr>
<tr>
<td>100 (max)</td>
<td>20987</td>
<td>7349826</td>
<td>417089</td>
<td>23914</td>
<td>18216</td>
<td>1182249</td>
</tr>
</tbody>
</table>

Table 7: Quantiles of Object Interarrival Times in the Test Programs. All times are measured in SPARC machine instructions.
Table 8: Quantiles of Object Holding Times in the Test Programs. All times are measured in SPARC machine instructions.

The quantile tables show the three distributions are greatly skewed. In particular, the median is a small fraction of the maximum in every case. As Jain points out [7], such a skew suggests that the median value should be used as a characterization of the central tendency of these distributions. As we have already seen, because a few small size classes dominate the size distributions of these programs, the quantile plot of object sizes is relatively uninteresting. The IAT quantiles show that the interarrival time between object allocations in the test programs is very frequent, the median time ranging from 100-1000 instructions. Likewise, most of the allocated objects are very short-lived, as the holding time quantiles show. The median object lifespan in the programs ranges from 90 to 26,000 cycles, with the 90th percentile less than 1 million cycles in all cases.

The final set of tables (Tables 9, 10, and 11) in this section present the ten most common classes observed in each of the three distributions. These tables illustrate that a small number of classes cover a large fraction of total observations in each of the distributions. From the percentages presented in Table 9, we see that the top ten sizes account for almost 90% of all object allocation in the six test programs. GAPSCRIPT has the most interesting size distribution with common objects of size 116 and 260 bytes. The other programs exhibit a very predictable distribution of object sizes with large fractions of the total objects less than or equal to 64 bytes in size.

The interarrival times indicated in Table 10 show that some programs are dominated by a single small interarrival time, on the order of 40-80 cycles. This regularity probably arises from an allocation occurring in a small, frequent loop. As the table shows, approximately 15-20% of all allocations are separated by this single, small IAT. Beyond the most frequent IAT, four of the six programs also captured almost 50% of the total distinct IAT's in the top ten. We see that the GAPSCRIPT program showed the least regularity of the programs measured.

Table 11 shows that the top ten holding times in each program represent a significant percentage of the total holding time distribution. GAWK, in particular, shows a great deal of regularity in object holding time: 62% of all objects lived either 86 or 89 cycles. Surprisingly, GAPSCRIPT, which shows
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>116</td>
<td>40796</td>
<td>37.58</td>
<td>37.58</td>
</tr>
<tr>
<td>2</td>
<td>260</td>
<td>40315</td>
<td>37.14</td>
<td>74.72</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>7708</td>
<td>7.10</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2854</td>
<td>2.63</td>
<td>84.45</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>2714</td>
<td>2.50</td>
<td>86.95</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>2415</td>
<td>2.22</td>
<td>89.18</td>
</tr>
<tr>
<td>7</td>
<td>26</td>
<td>1999</td>
<td>1.84</td>
<td>91.02</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>1551</td>
<td>1.43</td>
<td>92.45</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>1362</td>
<td>1.25</td>
<td>93.70</td>
</tr>
<tr>
<td>10</td>
<td>252</td>
<td>1357</td>
<td>1.25</td>
<td>94.95</td>
</tr>
</tbody>
</table>

**GHSTSCRS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
<td>20888</td>
<td>20.17</td>
<td>84.86</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>9667</td>
<td>9.63</td>
<td>94.49</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>4068</td>
<td>3.93</td>
<td>98.41</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>843</td>
<td>0.81</td>
<td>99.23</td>
</tr>
<tr>
<td>6</td>
<td>422</td>
<td>350</td>
<td>0.34</td>
<td>99.57</td>
</tr>
<tr>
<td>7</td>
<td>924</td>
<td>189</td>
<td>0.18</td>
<td>99.75</td>
</tr>
<tr>
<td>8</td>
<td>1692</td>
<td>142</td>
<td>0.14</td>
<td>99.89</td>
</tr>
<tr>
<td>9</td>
<td>1688</td>
<td>72</td>
<td>0.07</td>
<td>99.96</td>
</tr>
<tr>
<td>10</td>
<td>88</td>
<td>13</td>
<td>0.01</td>
<td>99.97</td>
</tr>
</tbody>
</table>

**Table 9:** Frequency of the 10 Most Common Size Classes in the Test Programs. All sizes are in bytes.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>69</td>
<td>8192</td>
<td>7.55</td>
<td>7.55</td>
</tr>
<tr>
<td>2</td>
<td>958</td>
<td>4078</td>
<td>3.76</td>
<td>11.30</td>
</tr>
<tr>
<td>3</td>
<td>1780</td>
<td>3649</td>
<td>3.36</td>
<td>14.67</td>
</tr>
<tr>
<td>4</td>
<td>989</td>
<td>2726</td>
<td>2.51</td>
<td>17.18</td>
</tr>
<tr>
<td>5</td>
<td>932</td>
<td>2635</td>
<td>2.43</td>
<td>19.60</td>
</tr>
<tr>
<td>6</td>
<td>1651</td>
<td>2457</td>
<td>2.26</td>
<td>21.87</td>
</tr>
<tr>
<td>7</td>
<td>1020</td>
<td>2208</td>
<td>2.03</td>
<td>23.90</td>
</tr>
<tr>
<td>8</td>
<td>2151</td>
<td>2035</td>
<td>1.87</td>
<td>25.78</td>
</tr>
<tr>
<td>9</td>
<td>1992</td>
<td>1709</td>
<td>1.57</td>
<td>27.35</td>
</tr>
<tr>
<td>10</td>
<td>748</td>
<td>1578</td>
<td>1.45</td>
<td>28.80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>122</td>
<td>5767</td>
<td>17.93</td>
<td>17.93</td>
</tr>
<tr>
<td>2</td>
<td>451</td>
<td>5203</td>
<td>16.18</td>
<td>34.11</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2774</td>
<td>8.62</td>
<td>42.73</td>
</tr>
<tr>
<td>4</td>
<td>1046</td>
<td>2510</td>
<td>7.80</td>
<td>50.53</td>
</tr>
<tr>
<td>5</td>
<td>224</td>
<td>888</td>
<td>2.76</td>
<td>53.30</td>
</tr>
<tr>
<td>6</td>
<td>454</td>
<td>576</td>
<td>1.79</td>
<td>55.09</td>
</tr>
<tr>
<td>7</td>
<td>203</td>
<td>376</td>
<td>1.17</td>
<td>56.26</td>
</tr>
<tr>
<td>8</td>
<td>181</td>
<td>374</td>
<td>1.16</td>
<td>57.42</td>
</tr>
<tr>
<td>9</td>
<td>119</td>
<td>373</td>
<td>1.16</td>
<td>58.58</td>
</tr>
<tr>
<td>10</td>
<td>1056</td>
<td>350</td>
<td>1.09</td>
<td>59.67</td>
</tr>
</tbody>
</table>

**Table 10:** Frequency of the 10 Most Common IAT Classes in the Test Programs. All times are measured in SPARC machine instructions.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>600</td>
<td>11915</td>
<td>5.25</td>
<td>5.25</td>
</tr>
<tr>
<td>2</td>
<td>391</td>
<td>11803</td>
<td>5.20</td>
<td>10.44</td>
</tr>
<tr>
<td>3</td>
<td>541</td>
<td>9122</td>
<td>4.02</td>
<td>14.46</td>
</tr>
<tr>
<td>4</td>
<td>713</td>
<td>8722</td>
<td>3.84</td>
<td>18.30</td>
</tr>
<tr>
<td>5</td>
<td>847</td>
<td>8534</td>
<td>3.76</td>
<td>22.06</td>
</tr>
<tr>
<td>6</td>
<td>168</td>
<td>6712</td>
<td>2.96</td>
<td>25.02</td>
</tr>
<tr>
<td>7</td>
<td>148</td>
<td>6472</td>
<td>2.85</td>
<td>27.87</td>
</tr>
<tr>
<td>8</td>
<td>542</td>
<td>3611</td>
<td>1.59</td>
<td>31.04</td>
</tr>
<tr>
<td>9</td>
<td>714</td>
<td>3607</td>
<td>1.59</td>
<td>32.62</td>
</tr>
<tr>
<td>10</td>
<td>538</td>
<td>3582</td>
<td>1.58</td>
<td>34.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86</td>
<td>12852</td>
<td>39.96</td>
<td>39.96</td>
</tr>
<tr>
<td>2</td>
<td>89</td>
<td>7178</td>
<td>22.32</td>
<td>62.27</td>
</tr>
<tr>
<td>3</td>
<td>197</td>
<td>887</td>
<td>2.76</td>
<td>65.03</td>
</tr>
<tr>
<td>4</td>
<td>158</td>
<td>374</td>
<td>1.16</td>
<td>66.19</td>
</tr>
<tr>
<td>5</td>
<td>411</td>
<td>340</td>
<td>1.06</td>
<td>67.25</td>
</tr>
<tr>
<td>6</td>
<td>464</td>
<td>102</td>
<td>0.32</td>
<td>67.57</td>
</tr>
<tr>
<td>7</td>
<td>485</td>
<td>102</td>
<td>0.32</td>
<td>67.88</td>
</tr>
<tr>
<td>8</td>
<td>408</td>
<td>102</td>
<td>0.32</td>
<td>68.20</td>
</tr>
<tr>
<td>9</td>
<td>429</td>
<td>102</td>
<td>0.32</td>
<td>68.52</td>
</tr>
<tr>
<td>10</td>
<td>540</td>
<td>95</td>
<td>0.30</td>
<td>68.81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37</td>
<td>3989</td>
<td>2.14</td>
<td>2.14</td>
</tr>
<tr>
<td>2</td>
<td>246</td>
<td>1329</td>
<td>0.71</td>
<td>2.85</td>
</tr>
<tr>
<td>3</td>
<td>375</td>
<td>1037</td>
<td>0.56</td>
<td>3.41</td>
</tr>
<tr>
<td>4</td>
<td>247</td>
<td>812</td>
<td>0.44</td>
<td>3.84</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>685</td>
<td>0.37</td>
<td>4.21</td>
</tr>
<tr>
<td>6</td>
<td>304</td>
<td>682</td>
<td>0.37</td>
<td>4.57</td>
</tr>
<tr>
<td>7</td>
<td>368</td>
<td>643</td>
<td>0.34</td>
<td>4.92</td>
</tr>
<tr>
<td>8</td>
<td>361</td>
<td>553</td>
<td>0.30</td>
<td>5.21</td>
</tr>
<tr>
<td>9</td>
<td>358</td>
<td>532</td>
<td>0.29</td>
<td>5.50</td>
</tr>
<tr>
<td>10</td>
<td>418</td>
<td>508</td>
<td>0.27</td>
<td>5.77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1570</td>
<td>1226</td>
<td>4.65</td>
<td>4.65</td>
</tr>
<tr>
<td>2</td>
<td>1375</td>
<td>1226</td>
<td>4.65</td>
<td>9.29</td>
</tr>
<tr>
<td>3</td>
<td>1705</td>
<td>1099</td>
<td>4.16</td>
<td>13.46</td>
</tr>
<tr>
<td>4</td>
<td>3880</td>
<td>1096</td>
<td>4.15</td>
<td>17.61</td>
</tr>
<tr>
<td>5</td>
<td>1318</td>
<td>1096</td>
<td>4.15</td>
<td>21.76</td>
</tr>
<tr>
<td>6</td>
<td>936</td>
<td>1095</td>
<td>4.15</td>
<td>25.91</td>
</tr>
<tr>
<td>7</td>
<td>1127</td>
<td>1095</td>
<td>4.15</td>
<td>30.06</td>
</tr>
<tr>
<td>8</td>
<td>6068</td>
<td>336</td>
<td>1.27</td>
<td>31.33</td>
</tr>
<tr>
<td>9</td>
<td>6263</td>
<td>336</td>
<td>1.27</td>
<td>32.61</td>
</tr>
<tr>
<td>10</td>
<td>2705</td>
<td>303</td>
<td>1.15</td>
<td>33.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>236</td>
<td>11239</td>
<td>10.35</td>
<td>10.35</td>
</tr>
<tr>
<td>2</td>
<td>267</td>
<td>7167</td>
<td>6.60</td>
<td>16.96</td>
</tr>
<tr>
<td>3</td>
<td>1225</td>
<td>6262</td>
<td>5.77</td>
<td>22.73</td>
</tr>
<tr>
<td>4</td>
<td>298</td>
<td>5870</td>
<td>5.41</td>
<td>28.13</td>
</tr>
<tr>
<td>5</td>
<td>594</td>
<td>3731</td>
<td>3.44</td>
<td>31.57</td>
</tr>
<tr>
<td>6</td>
<td>329</td>
<td>3550</td>
<td>3.27</td>
<td>34.84</td>
</tr>
<tr>
<td>7</td>
<td>965</td>
<td>2780</td>
<td>2.56</td>
<td>37.40</td>
</tr>
<tr>
<td>8</td>
<td>635</td>
<td>2581</td>
<td>2.38</td>
<td>39.78</td>
</tr>
<tr>
<td>9</td>
<td>794</td>
<td>2509</td>
<td>2.31</td>
<td>42.09</td>
</tr>
<tr>
<td>10</td>
<td>360</td>
<td>2501</td>
<td>2.30</td>
<td>44.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>3349</td>
<td>3.23</td>
<td>3.23</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>2332</td>
<td>2.25</td>
<td>5.49</td>
</tr>
<tr>
<td>3</td>
<td>46</td>
<td>1998</td>
<td>1.93</td>
<td>7.42</td>
</tr>
<tr>
<td>4</td>
<td>291</td>
<td>1685</td>
<td>1.63</td>
<td>9.04</td>
</tr>
<tr>
<td>5</td>
<td>115</td>
<td>1525</td>
<td>1.47</td>
<td>10.52</td>
</tr>
<tr>
<td>6</td>
<td>123</td>
<td>1510</td>
<td>1.46</td>
<td>11.97</td>
</tr>
<tr>
<td>7</td>
<td>1186</td>
<td>1251</td>
<td>1.21</td>
<td>13.18</td>
</tr>
<tr>
<td>8</td>
<td>352</td>
<td>1226</td>
<td>1.18</td>
<td>14.37</td>
</tr>
<tr>
<td>9</td>
<td>1125</td>
<td>1214</td>
<td>1.17</td>
<td>15.54</td>
</tr>
<tr>
<td>10</td>
<td>117</td>
<td>1086</td>
<td>1.05</td>
<td>16.59</td>
</tr>
</tbody>
</table>

**Table 11**: Frequency of the 10 Most Common HT Classes in the Test Programs. All times are measured in SPARC machine instructions.
less regularity in its IAT distribution, shows substantial regularity in its holding time distribution. It is also interesting to note that all of the holding times in this table are less than 10,000, again indicating that objects in all of these applications are most likely very short-lived. This result has implications for generation-based garbage collection algorithms that might be used to collect these objects [13].

4 Summary

We have presented data from one input data set for each of the six programs measured. Using the trace extraction and reduction techniques described, we have collected similar data from at least two inputs to each of these programs. We have also collected other statistical characterizations of the test programs and compared the accuracy of different synthetic models of program allocation in a companion paper [14]. All of the collected data has been formatted as a C program input file with declarations of the distributions of object size, holding time, etc. These C files can easily be compiled and linked with a program intended to manipulate the data. C files representing all inputs to all the programs described in this paper are publically available via anonymous FTP from ftp.cs.colorado.edu in the directory pub/cs/misc/MallocStudy. The compressed C files are labeled cfsmc-1,2,3,c.Z, etc. where each number represents a different input. In addition, a header file “DataHeader.h” is included that declares all the data structures defined in the .c files. Please feel free to use the data provided in this directory but we would appreciate your sending us e-mail ({zorn.grunwald}@cs.colorado.edu) indicating that you intend to use the data and how you intend to use it.

References

