Implementing Malloc: Students and Systems Programming

Brian P. Railing
Carnegie Mellon University
Pittsburgh, PA
bpr@cs.cmu.edu

Randal E. Bryant
Carnegie Mellon University
Pittsburgh, PA
randy.bryant@cs.cmu.edu

ABSTRACT

This work describes our experience in revising one of the major programming assignments for the second-year course Introduction to Computer Systems, in which students implement a version of the malloc memory allocator. The revisions involved fully supporting a 64-bit address space, promoting a more modern programming style, and creating a set of benchmarks and grading standards that provide an appropriate level of challenge.

With this revised assignment, students were able to implement more sophisticated allocators than they had in the past, and they also achieved higher performance on the related questions on the final exam.

KEYWORDS

malloc, programming assignment, systems programming

ACM Reference Format:
Brian P. Railing and Randal E. Bryant. 2018. Implementing Malloc: Students and Systems Programming. In *SIGCSE '18: SIGCSE '18: The 49th ACM Technical Symposium on Computer Science Education, February 21–24, 2018, Baltimore , MD, USA*. ACM, New York, NY, USA, 6 pages. https: //doi.org/10.1145/3159450.3159597

1 INTRODUCTION

Teaching Introduction to Computer Systems, a course developed at Carnegie Mellon University [2] and widely adopted elsewhere, exposes students to the basics of how the architecture, compiler, and operating system work together to support program execution. The course makes use of seven programming assignments, termed labs, that explore different aspects of the computer system. The most challenging lab for the course, distributed as part of the support materials for the Bryant-O'Hallaron textbook [3], requires students to implement a memory allocator based on malloc. Anecdotal reports by students, the instructors of more advanced systems courses, and subsequent employers of the students indicate that this lab provides them an important experience in low-level program implementation, debugging, and tuning.

In reflecting on the existing version of the malloc lab, we noted that high scoring student submissions required identifying particular tricks and knowledge of the testing environment, all the while not exploring the interesting design decisions and implementation work that we would have expected. Furthermore, when faced with the daunting design space during their work, students would often resort to changing and tweaking what they had done so far in hopes of following some random walk to a sufficiently optimized solution that would pass the assignment performance standards.

As a further shortcoming, students were encouraged to follow a programming style with extensive use of macros performing low-level address arithmetic. Many students struggled working with this style of programming—the compiler did not generate meaningful error messages, symbolic debugging tools could only provide a view into the post macro-expanded code, and it was very easy to make mistakes in address computations. A small, but significant number of students failed to write allocators that could even pass all of the correctness tests. The ability of modern compilers to generate efficient code via inline substitution has made most macro-based programming obsolete.

Previously, students would base their malloc on the code in The C Programming Language [6] and Computer Systems: A Programmer's Perspective [3]. From our review of student submissions, we concluded that there were several shortcomings in the assignment and developed four areas for revision to the programming lab. Listed here, these revisions will be discussed in greater detail later in this work:

- *Fully support a 64-bit address space.* This required creating a testing environment that allocated blocks of over 2^{60} bytes, even though no existing system has access to that much memory.
- *Promote a more modern programming style.* Rather than writing macros that performed low-level address calculations, students were required to make best use of the abstractions provided by C, while still achieving high degrees of memory utilization.
- *Use a set of carefully selected benchmark traces.* These benchmarks should challenge students to develop well-engineered designs that achieve an appropriate balance between memory utilization and throughput.
- *Follow a revised time line and grading standards.* We divided the assignment into two phases to help students better manage their time.

The rest of the paper is organized as follows: Section 2 provides the context of the course, Section 3 covers the overview of the malloc assignment, Section 4 explores the changes we made, Section 5 presents our reflections on the results, and Section 6 concludes this paper. Although the focus of this paper is on a specific assignment for a specific purpose, we believe that many of the factors addressed are important considerations for any systems programming assignment and that the approaches we devised have broader applications.

SIGCSE '18, February 21–24, 2018, Baltimore , MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159597

Figure 1: Memory layout of minimum-sized allocated and free blocks for an explicit-list allocator

2 COURSE BACKGROUND

This course is normally taken by Computer Science and Electrical and Computer Engineering majors in their second year, with a significant fraction of non-majors also enrolled, and about equal percentages of each of the three groups. All students are expected to have previous experience with C programming. The course is required for those majors, and the non-majors are usually taking the class to fulfill secondary major or minor requirements. This course also serves as the prerequisite for upper-level systems courses in computer science and engineering, including architecture, compilers, operating systems, and embedded systems. The malloc programming assignment is sixth, and most demanding, of seven assignments.

3 MALLOC PROGRAMMING LAB OVERVIEW

Students are required to implement the functions `malloc`, `realloc`, `calloc`, and `free`, following the API set in the C standard for each function. Their code is then linked into a driver that makes sequences of malloc and free calls. The driver populates allocated blocks with random data and verifies these bytes are preserved until freeing the block. Other correctness checks are also performed.

The lab exposes students to many aspects of systems programming including:

- Implementing library functionality given an API
- Exploring a design space with many choices in data structures and algorithms
- Understanding that heap memory is just an array of bytes that can be partitioned, reused, and redefined as a program executes
- Debugging and tuning code that relies on low-level data structure manipulation to meet targeted requirements
- Exploring possible trade offs between the conflicting goals of high memory utilization and high throughput

The following is a brief discussion of the implementation concerns present when working on this programming assignment. Wilson, et al. [10] has a detailed treatment of the design and evaluation of memory allocators.

Correct malloc implementations are measured with two performance metrics: throughput and utilization. Throughput is the number of operations completed in a measured amount of time. Utilization compares the heap space allocated to the program at the end of execution (the heap never shrinks) compared to the peak data requirement. The excess space required by the implementation is termed *fragmentation*, of which there are two types. Internal fragmentation occurs when the allocated block is larger than the payload, due to extra space required to manage the blocks and to satisfy alignment requirements. And as will be discussed later in Section 4.3, internal fragmentation is greater proportionately with

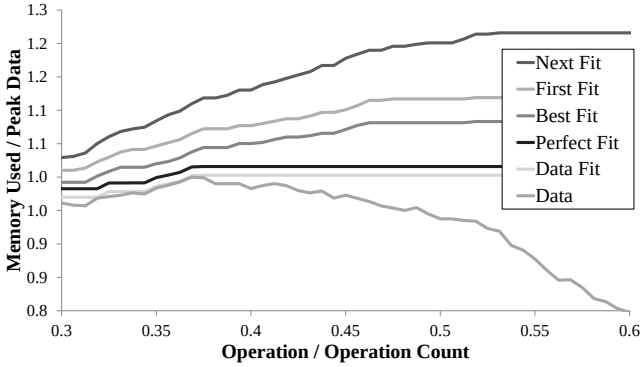

Figure 2: Memory usage for a synthetically generated trace for different free-block selection strategies. Usage is scaled relative to peak data requirement, and operations are scaled relative to the total number of operations

smaller requests. External fragmentation occurs when free blocks are available, but none are large enough to satisfy a request. Each fragmentation type is ameliorated by independent methods. Internal fragmentation is reduced by minimizing the overhead to track the allocated blocks, while external fragmentation is decreased by better selection of which free block to reuse. The fixed nature of pointers in C prevents any form of memory compaction, where allocated blocks are moved into a single region to reduce external fragmentation.

Student implementations start from code using an implicit free list, in which free blocks are located by traversing all allocated and unallocated blocks in memory order. Making the free list explicit with pointers between blocks significantly reduces the time to find whether a unallocated block can be reused. Both implicit and explicit list implementations use boundary tags for coalescing [7], with the layout for a minimum-sized block shown in Figure 1. Students can further improve the throughput of requests by segregating the explicit list based on sizes.

The free lists are searched on subsequent malloc requests to find if there are any blocks that satisfy the request. There are a variety of strategies, including: *next fit* that resumes searching where the previous search finished, *first fit* that selects the first block that could satisfy the request, and *best fit* that checks every block and selects the one closest to the requested size. Other design decisions can include whether to manage the free block list(s) by a FIFO, LIFO, address, or size ordering.

Figure 2 shows the memory usage for different free-block selection strategies focused on the peak, as the peak fragmentation characterizes the memory efficiency of the allocator. In addition to the block-selection strategies, this figure also includes two targets that serve as lower bounds for any allocator: *perfect fit* that assumes there is no external fragmentation, and *data fit* that assumes there is neither internal nor external fragmentation. In the figure, the fit curves continue to increase after the allocation peak, as the different strategies work to handle the churn of allocations and frees. Eventually, a sufficient percentage of memory is free that any new allocation in the trace can be satisfied and there is no further increase in memory usage.

One common optimization is to remove the footer from allocated blocks to allow larger payloads within a block. Other schemes can further improve utilization by special handling of smaller requests, or by reducing the headers for small blocks. Together, these optimizations are focused on reducing the internal fragmentation.

4 ASSIGNMENT REVISIONS

In the following section, we discuss our four revisions to the malloc implementation programming assignment.

4.1 64-bit Address Space

The previous version of the lab was derived from one based on a 32-bit address space. Students found they could exploit limitations of the testing framework. For example, they would deduce that no trace required more than 100MB of memory, and could therefore replace 64-bit pointers with 27-bit offsets and similarly for each block's size field. While these are valuable optimizations in a memory constrained environment, student submissions using this approach do not meet the programming lab objective of developing a general purpose memory allocator.

One possible solution would be to modify the traces to use larger request sizes; however, any such change would always have a known upper bound. Furthermore, students commonly run their code on shared servers, to which significant increases in memory demands could impact the shared platform. Indeed, there are no systems available that have access to the 1.8×10^{19} bytes that a true 64-bit machine could address.

We also felt it was important to provide a testing environment that would truly evaluate the ability of the student programs to handle a 64-bit address space, rather than relying on manual inspection by the students or the grading staff. There are many ways to inadvertently invoke 32-bit arithmetic in C, e.g., by using the expression `1 << n`, rather than `1L << n`, and so explicit testing is important.

To address both of these concerns, we developed an alternative testing infrastructure that would not be limited to the existing memory availability and could then permit testing of request sizes up to the maximum for the `size_t` parameter.

Using the LLVM compiler framework [8], a second, alternative binary `malloc-emulate` was built, where all of the memory operations (loads and stores) were modified to instead call an emulation library. This layer would either emulate the access if it was being made to the heap region or allow direct the access to system memory for other memory regions. The emulator exploits the fact that, even when a very large block is stored on the heap, the allocator need only access the bytes at the very beginning or the very end of the block (assuming no calls to `calloc` or `realloc`.) By emulating a very sparse address space, it can handle allocations of blocks containing 2^{60} or more bytes. With the emulated version, student code can then be tested to support full 64-bit pointers (going beyond current x86 support for 48-bits) and thereby determine if student code was robust against all access sizes or if it had been tuned to the smaller traces.

When running the emulated version, student code has roughly a 9× slowdown versus the native (non-emulated) implementation, and so throughput measurements are run on the native binary. It is possible for student code to detect whether it was being run with or without memory emulation, so the emulated tests are made using all of the native traces, and the memory utilization compared against the native execution. This prevents students from using the earlier optimizations during native runs and turning them off in emulation. In addition to rerunning the native traces, additional "giant" traces were prepared that require the emulation support to succeed.

Following the C standard [5], we use the implementation specified alignment for x86-64 [4] and require that student malloc implementations align the allocations to 16-byte boundaries.

```
/* Basic constants and macros */
#define WSIZE 8 /* Word and header/footer size (bytes) */
#define DSIZE 16 /* Double word size (bytes) */
/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))
/* Read and write a word at address p */
#define GET(p) (*(unsigned long *)(p))
#define PUT(p, val) (*(unsigned long *)(p) = (val))
/* Read the size field from address p. */
#define GET_SIZE(p) (GET(p) & ~0x7L)
/* Given block ptr bp, compute address of
 * its header and footer */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp))
                                        - DSIZE)
```

Figure 3: Old coding style based on macros (from textbook)

4.2 Modern Programming Style

Traditionally, C programming required using macros for common, low-level operations to avoid function call overhead. Also, the emphasis on carefully packing data to minimize memory usage, and the natural reuse of memory blocks in malloc implementations resulted in a significant usage of unstructured pointer arithmetic. This coding style is illustrated in Figure 3, showing excerpts from the allocator code described in the textbook. Many students would struggle when extending this code to support the necessary complexity to pass the assignment.

Modern C standards and compilers provide far greater support for types and performance, such that many of the motivations for macro code are obviated. We rewrote the starter code emphasizing two things: first, insofar as possible, operations should be made using explicit and appropriate type information, and second, inline and optimization decisions should be left to the compiler. Figure 4 is taken from the revised starter code provided to all students. Each internal block in the malloc implementation is now an explicit type; however, the footer cannot be included given that its location is dependent on the size of the payload. By compiling with optimization enabled, the function-based code has the same performance as macro-based code, and it significantly eases their debugging efforts when the optimizations are disabled.

When students make the first change to an explicit free list, they use `union` and `struct` to describe the types and positions of the required pointers. This contrasts with the earlier code where these fields would be known just by their offsets from the header, and

```
/* Basic declarations */
typedef uint64_t word_t;
static const size_t wsize = sizeof(word_t);
typedef struct block {
  /* Header contains size + allocation flag */
  word_t header;
  /* Placeholder for payload */
  char payload[0];
} block_t;
/* Pack size and allocation bit into single word */
static word_t pack(size_t size, bool alloc) {
  return size | alloc;
}
/* Extract size from packed field */
static size_t extract_size(word_t word) {
  return (word & ~(word_t) 0x7);
}
/* Get block size */
static size_t get_size(block_t *block) {
  return extract_size(block->header);
}
/* Set fields in block header */
static void write_header(block_t *block, size_t size,
                         bool alloc) {
  block->header = pack(size, alloc);
}
/* Set fields in block header */
static void write_footer(block_t *block, size_t size,
                         bool alloc) {
  word_t *footerp = (word_t *)((block->payload)
    + get_size(block) - dsize);
  *footerp = pack(size, alloc);
}
/* Locate start of block, given pointer to payload */
static block_t *payload_to_header(void *bp) {
  return (block_t *)(((char *)bp) -
                         offsetof(block_t, payload));
}
```

Figure 4: New coding style based on constants and functions

thereby reduces the chance that the student mixes the meanings of the offsets.

Even in making these changes to the baseline code, we did not do away with all macros. For example, macros are still provided to support assertions, debugging prints, and definitions of constant parameters. All student submissions were screened by a program that detects the use of function-like macros (those having arguments.)

4.3 New Traces

The selection of appropriate traces for the assignment can make a significant difference in establishing appropriate design decisions in the lab. For example, larger request sizes minimize the impact of internal fragmentation on reported utilization but can cause greater external fragmentation. A good allocator must overcome three challenges in achieving high utilization and throughput: (1) it must minimize internal fragmentation by reducing the overhead of the data structures, especially in the fields of the allocated blocks; (2) it must minimize external fragmentation, in which free blocks accumulate due to a poor block placement strategy; and (3) it must maximize throughput by minimizing the time required by each operation. A good set of benchmarks must stress all three of these aspects. No single benchmark can cover all of them, but hopefully a collection of well-chosen and well-designed benchmarks will clearly demonstrate the relative merits of different implementations.

Category	$p:q$	Unit	Maximum
Struct	70 : 30	8	256
String	60 : 40	1	256
Array	95 : 05	4	$\min(2^{18}, 10^9/N)$
Giant array	60 : 40	4	$2^{60}/N$

Figure 5: Power-law parameters for synthetic traces, when generating trace with N allocation operations

We created a new set of benchmark traces that would stress the diversity of program allocation behaviors. We did not conduct a systematic study of real program behaviors, rather we selected several programs with diverse allocation behaviors and also constructed additional synthetic traces to test specific categories of commonly allocated data.

We started with a set of programs and used the compile-time interpositioning method described in [3] to extract the sequence of calls to `malloc` and `free`. We selected sets of input data to these programs to generate a total of 12 traces having between 2,874 and 63,956 allocation requests, with an average of 34,515. Among the largest traces, one requires that the allocator manage up to 44,465 blocks at its peak, while another requires a total of 31.3 MB of payload data at its peak.

In addition, we generated a set of synthetic data that attempted to capture heap-allocated data structures storing structs, strings, and arrays:

Structs: These are used to implement nodes in lists, trees, and graphs. On 64-bit machines, their sizes will typically be multiples of eight to satisfy the alignment of embedded pointers

Strings: These will have no particular pattern of uniformity or size.

Arrays: These will have widely varying sizes, but they will be multiples of 4 or 8.

The block sizes within each of these categories were generated according to a power-law distribution, having a probability density function of the form $p(s) \propto s^{-\alpha}$, for some $\alpha \leq 1$. A power-law distribution can be characterized by a ratio of the form $p : q$, where $0 < p < 100$ and $q = 100 - p$, indicating that percentage p of the generated samples should be less than percentage q of the maximum value. For example, the familiar 80 : 20 rule states that 80% of the values are less than or equal to 20% of the maximum value. Each class is also characterized by a unit u, defining both the minimum value and a value for which all values must be multiples. Finally, each class has a maximum value M.

Figure 5 specifies the parameters for the different classes. Observe that both strings and structs have maximum sizes of 256 bytes, but strings have $u = 1$, while structs have $u = 8$. This difference has important implications for the internal fragmentation. Arrays have a maximum size of 262,144 (2^{18}) bytes, except when the trace has so many allocations N, that there is a risk that it will require a heap size of more than 100MB. In such cases, the maximum array

length is set to $10^9/N$. Normal arrays have an extreme power-law distribution 95 : 5, giving it a very long-tailed distribution. Finally, a class of giant arrays can be generated for traces to test the ability of the allocator to handle 64-bit sizes and pointers. These could have sizes up to $2^{60}/N$ ($2^{60} \approx 1.15 \times 10^{18}$), and with a fairly uniform 60 : 40 distribution. It should be noted that, although these parameters seem reasonable, we made no attempt to systematically evaluate the sizes of heap allocation requests in actual programs.

We generated four benchmark traces, each containing 40,000 allocation requests, for the regular benchmarks: one for each of the three categories, and a fourth containing allocations chosen randomly from the classes struct, string, and array, with probabilities 35%, 30%, and 35%, respectively. We also created a set of five "giant" trace benchmarks, generated allocations that require almost the entire 64-bit address space. The largest allocated block in these has a payload of 8.4×10^{16} (over 2^{56}) bytes.

The difficulty posed by a benchmark trace comes not just from how blocks are allocated, but also how they are freed. For example, a trace that only allocates blocks and never frees them (or frees them at the very end) will have almost no external fragmentation and so the free lists will be very short. These require only reducing internal fragmentation to get good performance. Additionally, block coalescing implies that a trace that frees most of the blocks at some point will end up with a small number of large free blocks. Such a configuration will not pose much challenge for the subsequent utilization or throughput.

We devised a simple scheme for inserting free operations into a trace (both the ones extracted from actual programs, as well as the synthetic traces) that proved successful in stressing both the throughput and the ability to minimize external fragmentation. Consider a trace containing N allocations that are never freed. At each such allocation step i, we generate a *free target* $t = 2i/N$, indicating a target number of free operations to insert after the allocation step. For $t < 1$, we flip a weighted (by t) coin to determine whether or not to insert a free operation. For $t \geq 1$, we repeatedly insert free operations and decrement t until $t < 1$, and then finish with a weighted-coin flip. When inserting a free operation, one of the existing blocks is selected at random to be freed. With this strategy, the program begins by allocating blocks, but freeing them only sporadically. At a midpoint, it approaches an equilibrium, allocating and freeing blocks in roughly equal portions. Then it increases the rate of freeing so that the expected number of allocated blocks at the end should be nearly 0. We also free any allocated blocks at the end to complete the trace.

Figure 2 illustrates the performance of allocators following different block placement policies near the peak for one of the synthetic benchmarks. This is that point where external fragmentation is at its maximum, challenging both utilization and throughput.

We rejected constructing any adversarial traces that might unduly penalize specific design decisions, such as one that would force the allocator's search strategy to fully traverse a FIFO list. The only defense against such traces would be to implement an allocator with strong worst-case performance guarantees, e.g., using balanced tree data structure to implement the free list. Such a requirement was deemed beyond the scope of the course. We also worked to avoid any benchmark trace that would target particular implementation characteristics. Finally, in keeping with the previous lab assignment, traces using `realloc` are tested only for correctness and not any performance metrics.

Semester	Version	Assignment Score	Exam Scores
Spring 2016	Old	78.0	78.0
Fall 2016	New	75.1	87.0
Spring 2017	New	75.3	89.0

Figure 6: Malloc Assignment and final exam question scores (both with maximum values of 100), for the old and new versions of the assignments

4.4 Grading Standards

Having established a set of benchmarks, existing norms for student submissions no longer apply. Instead we performed an extensive benchmarking process, implementing a number of allocators trying different strategies for reducing internal and external fragmentation. We measured how each performed in terms of throughput and utilization, and then selected cutoff points based on the features we considered achievable by all students versus those of increasing difficulty and complexity. Students implementing segregated free lists, removing footers, and an approximation of best fit would earn a B (around 85%) for the assignment. These features take students a self-reported average of 20–30 hours to complete, which fits with a multiweek assignment. Students completing those features faster generally use additional time to make further improvements to their submissions.

Furthermore, we found that splitting the assignment into two parts by introducing a checkpoint also aided student learning and experience. Given the sequence of features to implement, transitioning the baseline code from implicit free lists to explicit, segregated free lists roughly corresponded to half of student time, as they built up their understanding of the code and the approaches required to debug their implementation. This transition only significantly impacts the throughput of the implementation, which allows students to focus on optimizing for a single constraint. The second part of the assignment requires them to then improve utilization, mostly by reducing internal fragmentation.

5 REFLECTIONS

A major question in any update to course material is whether student learning is improved. As we wrote new traces and grading standards for the assignment, we cannot easily compare before and after the revision on the assignment itself. However, one set of questions on the final exam test students' knowledge of malloc. In Figure 6, we compared the before (Spring 2016) scores with those after revising the lab (Fall 2016 and Spring 2017), as well as including the programming assignment scores for comparison.[1] Using an independent samples t-test, we found no statistically significant difference in the assignment scores, yet a clear difference (p value less than 0.01) in the related exam scores before and after the lab revision. However, the score distribution on the assignment is such that only half of the students in each semester are earning a B.

This assignment gave us considerable exposure to students' efforts to debug their implementations. In particular, corruptions in the internal state of the implementation will not immediately crash

[1]Due to other course changes, we cannot make a broader comparison between semesters.

```
(*(void **)((*(void **)(bp)) + DSIZE)) =
                            (*(void **)(bp + DSIZE));
```

Figure 7: Untyped malloc implementation

```
bp->prev->next = bp->next;
```

Figure 8: Typed malloc implementation

the allocator, but rather instead fail at some later point. Some failures are crashes (segmentation faults or bus errors), but many others are observed as overlapping allocated blocks or corruption of an allocated block's payload. We emphasize the use of gdb, specifically using the two recitations during the assignment to work through debugging techniques using several simple implementations of malloc that have bugs commonly seen by students; however, as observed by others [1] [9], many students will use other debugging approaches. For example, we require students to write a function, the heap checker, that will check appropriate invariants of their implementations. And finally, many students debug their code by manually tracing the state or introducing print statements to observe the state.

Given the design space described in Section 3, this assignment is often one of students' first exposure to a problem with an open design space. Regularly, students attempt to glean specific insights from the instructional staff such as "How many segregated free lists should my implementation use?" or "Now that I have implemented X, what should I do next?". We emphasize the need to test and explore the options available in the potential design space, while providing a basic set of possible features. They both enjoy the ownership that comes from making these decisions, while also expressing frustration with the lack of specific guidance. The guidance is improving as teaching assistants come through the course with the new lab. And for those TAs with experience across both versions of the assignment, they report a greater satisfaction and ease of helping students from this revision.

From the grading and support side, it can be difficult to understand the meaning of purely pointer-based code. Figure 7 provides one extreme example of the minimal level of type information that student code may use. When students ask for assistance debugging similar code, our common response has been to recommend a type-based rewrite, resulting in code similar to Figure 8. And for many, this rewrite is sufficient to find the cause of the bug and fix the original issue.

However, this typed code is not without cost. Those students extending the baseline code with additional pointer casts can violate strict aliasing rules, such as by casting `block_t*` (see Figure 4) directly to `free_block_t*` (or other alternative structure). This cast can have undefined behavior in the C standard, whereas previously all pointers would be `char*` or `void*` and not have this behavior. Identifying undefined behavior that is causing failures is tricky and students can be particularly frustrated that introducing print statements to debug the failing behavior can disrupt the compiler's analysis and optimizations, such that this and other undefined behaviors are no longer failing. Finally, a small percentage of students state a preference for the traditional, macro-based code.

6 CONCLUSION AND FUTURE WORK

This paper has shown our revisions to a demanding systems programming assignment. These changes have helped refocus student effort on the learning objectives for the assignment. And our preliminary analysis of student scores shows a significant improvement, indicating that students are learning the material better than before.

There are several opportunities to continue the assignment rewrite. First, we can revisit the trace generation of Section 4.3 and compare it against a more systematic study of allocation patterns. Second, the throughput measurements can be sensitive to machine configuration and load, so we are investigating how to maintain the time component while minimizing the sources of variation. Finally, we are continuing to update our guidance to students on both how to improve their design and style, as well as exploring how to better prepare students for debugging complex and optimized code.

ACKNOWLEDGMENTS

Matthew Salim, an undergraduate student at CMU, participated in many parts of the lab development. The CMU Eberly Center for Teaching Excellence has provided helpful guidance in how best to assess the effectiveness of the assignment in enhancing student learning, as well as assisting with the collection and analysis of student scores. The National Science Foundation, under grant 1245735, has supported some of our efforts in course material development.

REFERENCES

[1] Basma S. Alqadi and Jonathan I. Maletic. 2017. An Empirical Study of Debugging Patterns Among Novices Programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 15–20. https://doi.org/10.1145/3017680.3017761

[2] Randal E. Bryant and David R. O'Hallaron. 2001. Introducing Computer Systems from a Programmer's Perspective. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. ACM, New York, NY, USA, 90–94. https://doi.org/10.1145/364447.364549

[3] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson.

[4] Intel Corporation, Santa Clara, CA 2017. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, Santa Clara, CA. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[5] ISO/IEC 9899:2011 WG14 2011. *Programming Languages - C (C11)*. ISO/IEC 9899:2011 WG14. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[6] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA.

[7] Donald E Knuth. 1973. *Fundamental algorithms: the art of computer programming*.

[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–88. http://dl.acm.org/citation.cfm?id=977395.977673

[9] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 163–167. https://doi.org/10.1145/1352135.1352191

[10] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*. Springer-Verlag, London, UK, UK, 1–116. http://dl.acm.org/citation.cfm?id=645647.664690