An empirical study of a novel technique: rationed-memory compiling

by

Ioan Tudor Leu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

May 19, 2005

Certified by

Martin Rinard
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
An empirical study of a novel technique: rationed-memory compiling

by

Ioan Tudor Leu

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 2005, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

We present *rationed-memory compiling*, a new technique designed to reduce the memory consumption of programs. The technique lowers memory usage by decreasing the amount of memory that programs allocate, but not use actively. With rationed-memory compiling, whenever a program requests a memory block, the compiler returns a block smaller than the requested size. The compiler handles subsequent reads and writes to the memory block as usual if the accessed location is within the allocated block. If the accessed location is outside the allocated block, the compiler redirects the access to a hash table which stores and indexes out-of-bounds writes and returns the stored values on out-of-bounds reads from corresponding locations. For programs that over-allocate memory, the compiler yields savings because it allocates smaller memory blocks, while it also provides backup storage in the hash table for the writes outside the allocated blocks.

We developed a C compiler that implements the rationed-memory technique and tested it on a series of programs that use a diverse range of data structures. Our tests show that the rationed-memory technique is very effective in reducing the memory usage of data structures implemented using arrays or buffers, such as stacks or queues. Furthermore, our study presents cases of over-allocation of memory in open-source applications, supporting our claim that the rationed-memory technique is a versatile tool with a great potential of reducing the memory consumption of programs.

Thesis Supervisor: Martin Rinard
Title: Associate Professor
Acknowledgments

First and foremost, I would like to thank my thesis advisor, Martin Rinard, to whom I owe the research idea driving this thesis and whose enthusiastic mentoring and encouraging approach made my graduate year a rewarding and fun learning experience. More important than the technical knowledge I have accumulated from him during my work on this project, I will take with me at least three key pieces of advice to always guide me in my career: find your competitive edge, challenge the status quo, and always be brilliant.

I am grateful to Cristian Cadar, Daniel Roy and Daniel Dumitran, who worked on the boundless memory blocks project on which I based my implementation of the rationed-memory compiler. Their efforts were essential to the success of my work. I also thank Alexandr Andoni, Mihai Pătraşcu and Alexandru Sâlcianu for fruitful technical discussions, many good ideas, and for being smart computer scientists.

Looking back on my school years, there are many great teachers who inspired and guided my interest in science and engineering. I would like to thank Patrick Winston, who stimulated my interest in understanding human intelligence and also taught me the importance of good communication skills. I also owe the deepest thanks to Emil Onea and Dana Lica, whose dedication and support as my teachers cannot be quantified, and without whose help I would have never started on my journey to MIT.

My time here would have been infinitely harder without the help of my family. Even from thousands of kilometers away, their love and support kept me going. Thank you.

I would like to thank Alex, a brave roommate, dependable friend, and faithful comrade in many nights spent in Athena clusters writing our theses side by side. (Speaking of Athena, I am deeply attached to w20-575-82..86.mit.edu and m12-182-16.mit.edu, on which I wrote most of this thesis) My gratitude also goes to my Romanian friends at MIT, my Romanian friends in Romania, and my friends from EC 2E. And finally, thanks to Jay Scheib and Jane Lane for artistic inspiration.
Contents

1 Introduction ............................................................... 13
  1.1 Vision .............................................................. 13
  1.2 Background and motivation ....................................... 13
  1.3 The rationed-memory compiling technique ...................... 15
  1.4 Advantages and drawbacks ....................................... 16
  1.5 Contributions ..................................................... 18

2 Implementation .......................................................... 19
  2.1 Managing memory objects information for bounds checking in the CRED compiler ...................... 20
  2.2 Out-of-bounds access handling with boundless memory blocks ................................................. 21
  2.3 Rationing memory .................................................. 22

3 Experience ................................................................. 25
  3.1 Over-allocation in open source programs ....................... 26
  3.2 Running programs with rationed-memory compiling ...................... 28
      3.2.1 Breadth-first graph search ................................ 29
      3.2.2 Convex hull .................................................. 31
      3.2.3 Single source shortest paths .............................. 32
      3.2.4 Eratosthenes’ prime number sieve ....................... 33
  3.3 Discussion .......................................................... 34
4 Future work

4.1 Compiler functionality upgrades ............................................. 37
4.2 Ideas for enhancement .......................................................... 39

5 Related work

5.1 Data size optimizations for object-oriented programs .................. 43
5.2 Eliminating equivalent object instances and dead data members ....... 44
5.3 Detecting and exploiting narrow bitwidth computations ................. 44

6 Conclusion

A Over-allocation in Pine 4.44 – source code ................................. 49

B Programs used in experiments ................................................... 53

B.1 Breadth-first search .............................................................. 53
B.2 Convex hull .......................................................................... 55
B.3 Single source shortest paths .................................................... 57
B.4 Eratosthenes’ prime number sieve ........................................... 59
List of Figures

2-1 Allocation of memory with the rationed-memory compiler and redirection of reads and writes. .......................... 23

3-1 Breadth-first search with the rationed-memory technique .................. 30
### List of Tables

3.1 Sample test results for the convex hull program compiled with the rationed-memory compiler .............................................. 32

3.2 Test results for Erathostene’s prime number sieve compiled with the rationed-memory compiler ............................................. 34
Chapter 1

Introduction

1.1 Vision

If we are to build better software, we need to enable programs to use judiciously the resources available in a computing system. One of the most important system resources is memory, and our goal is to explore alternative techniques of handling memory that enable us to avoid wasteful use caused by traditional memory management methods. This thesis presents a novel technique, rationed-memory compiling, which is designed to lower the memory consumption of programs by reducing the amount of memory allocated but not actively used by programs.

1.2 Background and motivation

The memory management framework in many programming languages, such as C, C++, Pascal or Fortran, for example, allows programmers to reserve memory blocks to hold entities such as arrays, buffers or structs. When using these languages, programmers can reserve memory blocks of chosen fixed sizes through dynamic or static allocation and then use the blocks at their own will. Even a language such as Java, which is designed to take memory management control away from the user, allows programmers to dynamically allocate arrays of chosen sizes.

When operating under this framework, programmers usually reserve as much
memory as they think their program will require. Good programmers will try to determine a priori the worst-case scenario of use for a memory block and produce an accurate estimate of the maximum required size of that memory block. There are cases when it is very difficult or even impossible to identify a worst-case scenario, leaving even the best programmers guessing how big of a memory block they should request. In such cases, programmers will usually err on the side of caution, allocating a large memory block and trying to minimize the risk of exposure to buffer overflows and similar memory errors.

Whether it is easy or hard to decide how much memory to allocate for an entity in a program, it is certain that when the program receives the requested memory block, it has exclusive control over that entire block, regardless of how much or how little it uses. Even if it does not reach frequently the worst-case scenario where it accesses the entire memory block for its computations, the program still consumes the memory and makes it unavailable to other processes in the system. On a larger scale, if we consider a complex application employing various types of data structures, it is highly unlikely that all the data structures will reach their worst-case usage scenario during the same program execution stage. This implies that the amount of memory actively used and needed by a program throughout its execution is less than the total amount of memory allocated by that program.

The difference between the memory allocated by a program and the memory active in computations during the program execution translates into a great potential for savings in the memory usage of that program. However, traditional compilers and memory management approaches cannot harness this potential, because they do not provide enough flexibility for programs to adjust their memory requirements at runtime. Even if programmers use the standard allocation and re-allocation techniques to adapt the memory usage of their programs during the execution, these attempts increase the burden of memory management on the programmer side, causing increased complexity in programs and possible unwanted side effects.
1.3 The rationed-memory compiling technique

We introduce a new technique called *rationed-memory compiling* to reduce program memory consumption by taking advantage of the memory left unused by programs during their execution. Whenever a program requests a memory block, the compiler allocates a memory block whose size is a fraction of the requested size, and returns back its address. Then, program reads or writes to the memory block occur in the following way: if the desired access location falls within the allocated memory block, the access proceeds as usual. If, on the other hand, the desired access location is past the boundary of the allocated block, the compiler redirects the access to a hash for out-of-bounds accesses. This hash table stores the values of out-of-bounds writes indexed under the intended write address. Out-of-bounds writes update entries in the table or create new entries, while out-of-bounds reads return the corresponding stored value, or a default value for uninitialized addresses.

The technique produces memory savings when programs use only parts of the memory blocks they request. The savings arise from the difference between the requested size of memory blocks and the size of the allocated blocks. Accesses past the boundaries of allocated memory blocks use memory in the hash for out-of-bounds accesses, but if such accesses are rare, the memory consumption of the hash is low. Our study of open-source programs and commonly used algorithms suggests that many types of data structures used in programs have a low memory usage ratio on common cases, and that programmers customarily over-allocate memory to hold various program entities.

With our technique, the requested size of a memory block becomes an indication of the maximum amount of memory that the corresponding data structure may use, rather than a firm limit. Moreover, our technique shares an important property with the boundless memory blocks technique [8]: conceptually, each memory block has unbounded size. Therefore, if the estimation of the worst-case scenario is wrong and the program needs more memory for a certain data structure, the compiler can supply that extra memory.
We developed a C compiler that implements the rationed-memory technique and used it to compile a series of implementations of commonly-used algorithms and data structures. Our results show significant reductions in memory usage by those programs, and prove the versatility of the technique on data structures with varied patterns of memory use. While we have not tested our compiler with larger applications yet, we have gathered examples of over-allocation in open-source C programs, showing the potential of our technique to reduce memory consumption in such programs.

1.4 Advantages and drawbacks

Among the techniques for reducing the memory consumption of programs, rationed-memory compiling presents several benefits:

- **Versatility:** Rationed-memory compiling produces reductions of memory consumption for programs with different patterns of memory use. The technique is able to exploit over-allocation in arrays or buffers, as well as low memory usage of a variety of data structures implemented using arrays or buffers.

- **No additional memory management burden:** To take advantage of the reductions produced by our technique, developers do not need to change the way they manage the memory used by their programs. Programs request the allocation of memory blocks as usual, and the compiler handles all the mechanisms that implement the technique without programmer intervention during the execution of the program.

- **Compatibility:** To adopt the technique, programmers do not need to use new libraries of code or a different programming language, nor do they need to modify their programs. Recompiling the programs with a rationed-memory compiler suffices.
Some potential drawbacks are:

- **Possible performance overhead:** The memory reads or writes that get redirected to the hash for out-of-bounds accesses will cause a performance overhead, because they are slower than direct accesses to memory. However, knowledgeable choices of compiler parameters that avoid placing a large duty on the hash should alleviate this problem.

- **Calibration required:** To maximize the reductions in memory consumption produced by the technique, developers need to calibrate the parameters of the rationed-memory compiler. The amount of memory saved depends both on the compiler parameters — the ratio of allocated versus requested memory and the size of the hash for out-of-bounds accesses — and on how the program uses the memory it requests, and the developer has to find the optimal fit between these two factors. However, even if the developer does not perform a precise calibration of the compiler, there are simple conservative choices of parameters which guarantee a low performance overhead and produce moderate memory savings if the program allows it; one such setting would be a high ratio of allocated versus requested memory, and a hash of moderate size.

- **Extra memory required in some cases:** If the program allocates memory in a very precise manner, meaning that it uses all or nearly all the memory it requests, compiling the program with the rationed-memory technique will cause an increase in the memory usage of the program. This is due to the fact that the the hash for out-of-bounds accesses holds both the written values and the intended write addresses. Therefore, in this memory usage scenario the rationed-memory technique is not effective. However, because the compiler can log how the hash is used, it easy to detect such a scenario and determine whether it is appropriate to use the technique in this case.
1.5 Contributions

This thesis makes the following contributions:

- **Rationed-memory compiling:** We present the novel technique of rationed-memory compiling, in which the compiler allocates smaller memory blocks than requested by programs and redirects accesses past the boundary of the allocated blocks to a special hash table for out-of-bounds accesses, with the purpose of reducing the program memory consumption.

- **Implementation:** We developed a C compiler that implements the technique and is able to apply it on programs written in standard C that use dynamically allocated arrays and buffers.

- **Evaluation:** We evaluated the potential of the rationed-memory technique to produce memory savings in programs and we tested our compiler implementation on a series of common algorithms. Our results show that the technique can greatly reduce the amount of memory used by a variety of data structures commonly used by programs.
Chapter 2

Implementation

The implementation of the rationed-memory compiler (RMC) is based on the boundless memory blocks (BMB) compiler [8], which in turn builds on the CRED compiler developed by Ruwase and Lam [11]. As a safe-C compiler, CRED performs checks to detect out-of-bounds memory accesses in programs, using supplementary information about the memory blocks used in the programs. Whenever an out-of-bounds memory access occurs, CRED halts the program and identifies the error.

The boundless memory blocks compiler uses a different strategy in dealing with out-of-bounds accesses: it allows programs to continue their execution without throwing an exception or corrupting other data structures. The BMB compiler makes continued execution possible by storing out-of-bounds writes in a table and then returning the respective values when out-of-bounds reads occur at the same addresses as the writes.

Besides providing immunity to buffer overflows and other memory errors, the boundless memory blocks technique presents another interesting property: conceptually, memory blocks have unbounded size in programs compiled with the BMB compiler. The rationed memory compiler employs exactly this property to modify the memory usage of a program. Namely, whenever a program allocates a block of memory, the compiler gives it back a block smaller than requested, then handles all accesses beyond the allocated size using the table mentioned above.

In the remainder of this chapter, we will describe how the CRED compiler main-
tains supplementary information about the memory blocks used by programs, how the boundless memory blocks technique uses this information and extends the capability of the compiler to allow for out-of-bounds accesses to memory without interrupting execution, and finally, how the rationed memory compiler builds on these capabilities to allow programs to run using less memory than requested.

2.1 Managing memory objects information for bounds checking in the CRED compiler


To identify pointer references as in-bounds or out-of-bounds, Jones and Kelly’s scheme uses an object table, which keeps track of the base address and the size of each static, heap and stack object. Obviously, the base pointer of an object is always in-bounds. For pointers that are derived through pointer arithmetic from a starting pointer (for example, a base pointer or some other pointer), the decision of whether they are in-bounds or out-of-bounds depends on their referent object, which is the object referred by the starting pointer. The newly computed pointer is in-bounds if and only if it points into the same referent object as the starting pointer.

Jones and Kelly’s scheme does not allow for the use of out-of-bounds pointers, except for the one which points to the next byte after its intended referent object; this particular out-of-bounds pointer could be used for example as a limit pointer when looping through an array. However, the scheme rejects another type of referencing that is common in C programs – computing with pointer arithmetic a pointer past the bounds of an object in memory, then using pointer arithmetic again to return into the bounds of that object.

Ruwase and Lam remove this limitation in CRED using out-of-bounds objects (OOBs), which are generated for each out-of-bounds pointer. An OOB links an out-
of-bounds pointer to its intended referent object or data unit, keeping track of the base address of the data unit and of the out-of-bounds address. The OOBs are not stored in the object table, but rather in a separate out-of-bounds object hash table. Because it remembers the offset of the out-of-bounds pointers, CRED can translate pointers derived from out-of-bounds pointers back into regular in-bounds pointers. Besides their role in translating pointers, OOBs turn out to be extremely useful in handling out-of-bounds reads and writes, as shown in the following section.

2.2 Out-of-bounds access handling with boundless memory blocks

So far, we have seen that CRED handles out-of-bounds pointers created during the execution of a program, but it stops the execution of the program whenever such a pointer is being referenced. The boundless memory blocks (BMB) compiler [8] presents a different philosophical perspective on accesses to out-of-bounds memory locations: they should be regarded as normal, though uncommon, events in the execution of a program, and the program should continue running after such an access; the compiler saves the information about the out-of-bounds access into a log that the programmer may then use to eliminate the access or increase the size of the memory block in question.

However, continuing the execution does not mean that the an out-of-bounds write actually occurs at the specified out-of-bounds location. Instead, the BMB compiler stores the written value in a hash table indexed under the memory block and offset of the write. When the program attempts to read from an out-of-bounds location, the compiler first looks into the hash table. If there was a previous write to the same offset and the same memory block, then the compiler retrieves the value saved under that index in the hash table and returns it. Otherwise, it returns a default value.

The hash table for out-of-bounds writes has a fixed size in the BMB implementation, in order to avoid memory leaks and to eliminate the risk of denial of service.
attacks, and it uses a least recently used replacement policy.

## 2.3 Rationing memory

Our compiler takes advantage of the infrastructure provided by CRED and the BMB compiler to maintain the stability of running programs after enforcing the rationed-memory approach.

The compiler adjusts the memory usage of a program in the following way: whenever a call to `malloc` occurs during the execution, the return value is a pointer to an allocated memory block that is a fraction of the requested size. This reduction is applied for all `malloc` calls, except the ones for small blocks of memory (in the current implementation, blocks smaller than 50 bytes), the reason for this exception being that the program is likely to use the requested small memory blocks entirely, and indirection through the hash table would be unnecessary. Further, the reduction in memory usage for these blocks would be insignificant.

The change to `malloc` brings a modification in the object table described in section 2.1: for each memory object, we now keep track of its base address, its size and its *requested size*. In the case of reduced memory objects, the size in the object table is the *allocated size*, which is smaller than the requested size. (The requested size field is now used for debugging and profiling purposes; a different version of the rationed-memory technique could use it to adjust dynamically the amount of allocated memory – see chapter 4)

The out-of-bounds objects representation, as well as the hash table for out-of-bounds writes, do not change from what is described in section 2.1.

Accesses to memory take place in the same way with rationed-memory compiling as with boundless memory blocks, only in this case the reads and writes past the allocated memory block are treated as out-of-bounds reads and writes, even if in the case of normal memory allocation they would have occurred within bounds. This follows from the way the rationed-memory compiler allocates the memory blocks initially. A program compiled with the compiler references and accesses all the addresses past the
char* a = malloc (1000);
char* b = a + 800;
*b = 3;

Figure 2-1: The rationed-memory compiler allocates a smaller memory block than the program requests. The compiler redirects writes beyond the allocated zone to the out-of-bounds writes hash table, and subsequent reads from that address will retrieve the value from the table.

allocated memory block using the hash table for out-of-bounds writes. This means that the program does not use memory for any of the unreferenced addresses past the allocated memory block. Thus, any unreferenced addresses with an offset from the base address larger than the allocated size, but smaller than the requested size, contribute to the savings in memory use.
Chapter 3

Experience

In describing our experience with rationed-memory compiling, we aim for two goals. The first goal is to show how memory over-allocation occurs in a variety of scenarios when writing programs, and how the rationed-memory compiling approach helps in decreasing the memory waste in these scenarios. Based on that, we argue that rationed-memory compiling is a useful and powerful technique of reducing memory usage in programs. The second goal of this chapter is to provide several examples of programs compiled with the current implementation of the rationed-memory compiler, and show how our novel compiling technique works in practice.

This chapter is structured as follows: we begin by showing examples of over-allocation in Pine, the well-known open-source mail client, indicating how over-allocation occurs in widely used software. We also argue that rationed-memory compiling would not impact the reliability and availability of this program. In the following section, we display examples of over-allocation in commonly used algorithms and data structures, and present the results and the behavior of their implementations when compiled with the rationed-memory compiler. We wrap up with a discussion of our results.
3.1 Over-allocation in open source programs

Over-allocation happens frequently in programs, and it is not necessarily the result of negligence on the part of programmers, but rather it is due to programmers being cautious about having sufficient memory to fit the various entities in their code. When writing code, it is often hard or even impossible to judge precisely the maximum possible size of some input, of a string buffer used to keep some intermediate computation results, or of some particular data structure used in the program. The historical approach to these cases has been to allocate memory for the worst-case scenario, ensuring that in the most unfavorable circumstances the program is able to fit all data into memory.

However, this approach has its own disadvantages. First, it is hard in many cases to estimate correctly even the worst-case scenario in the situations just described, as we will see in several examples in this chapter. The estimate may be too generous, leading to excessive memory consumption, or it may actually be too tight, in which case the program either would not be able to fit all the data into memory, or it would crash while overrunning the boundaries of some allocated memory block. Second, even when the programmer assesses correctly the worst-case scenario, that does not prevent the waste of memory in all the instances when the program does not reach that scenario.

While implementing the rationed-memory compiler, we have looked at various open-source programs in order to try to gauge the impact of the rationed-memory technique on such programs. We ran across several instances of over-allocation in version 4.44 of Pine, the popular email client, and two of them illustrate some of the issues we just discussed. (We show the relevant pieces of source code related to these instances in appendix A.)

The first instance of over-allocation occurs when Pine creates a full-form To field for outgoing messages starting from what the user provides as input in the message editing window. For example, if the user typed in:

To:  steve, alyssa, bill@microsoft.com,
Pine would expand that to:

To: Steve Jobs <steve.jobs@apple.com>, alyssa@mit.edu,
"Bill Gates" <bill@microsoft.com>

using the user’s address book and the local domain name.

Pine expands the To field into a string buffer allocated dynamically. The requested buffer size is the constant MAX_ADDR_FIELD, arbitrarily set to 10000. This indicates that the programmer could not provide an accurate estimate of the maximum size of the To field, and decided to lean on the side of caution by requesting a large amount of memory for the string buffer.

Indeed, it is hard to decide on a bound for the maximum size of the To field, but it is clear that for nearly all possible messages a smaller buffer would be able to hold the field, even in its expanded form. Further, the unlikely, but possible case where the expanded field is larger than 10000 bytes may threaten Pine’s functionality, as the program is likely to crash.

Rationed-memory compiling would solve the problem of over-allocation, even if the code containing the allocation request is left as it is. When Pine would request a large string buffer, the compiler would provide just a fraction of the requested memory. For most cases, the To field would still fit in the buffer, reducing greatly the memory usage. In the cases where the field would expand beyond the allocated buffer, the hash for out-of-bounds writes would be able to hold the extra bytes. However, no more memory than it is necessary would be used.

Another example of over-allocation takes place when Pine parses the From fields of messages in the INBOX folder, in order to display them in the message index. For each message, Pine allocates a string buffer in which it copies the contents of the From field, paying attention to the special characters ‘‘ and \, The parser does not copy those characters directly to the buffer, but instead it expands them to escape sequences consisting of two characters: \’’ and \, respectively.

From the description of this parsing procedure, it should be easy to see that in the worst case, all characters in the From field may be special characters, and the destination string buffer should be twice as large as the original field in that case, if we
were to use a standard compiler. However, despite being aware of the fact that some characters may be expanded to two-character sequences, the programmer still does not allocate enough memory for the destination string buffer, which constitutes a buffer overflow vulnerability documented in [1]. This is an example of how programmers can err in estimating the maximum possible size for a memory block that they allocate. If this piece of code were compiled with the rationed-memory compiler, not only would that reduce the memory waste for all the times when the From field would not have many special characters, but also the buffer overflow vulnerability would disappear. The rationed-memory compiling technique inherits this improvement in security from the boundless memory blocks technique described in [8].

3.2 Running programs with rationed-memory compiling

This section presents the results of experiments we performed on programs compiled with the current implementation of the rationed-memory compiler. We successfully compiled and ran implementations of several well-known algorithms with our compiler and while doing so, we gathered practical evidence of how the compiler decreases memory usage for commonly used data structures. It should be noted that in our test programs we allocated memory for data structures in a traditional, conservative fashion, simply requiring enough memory for the worst-case scenario. We did not use any fancy techniques to save memory inside the programs, nor did we allocate more than what would have been required in the worst case. Our purpose for the experiments was to show the potential of the rationed-memory technique to reduce memory usage across a variety of programs, regardless of particular memory management tricks which could apply to one program in particular.

Note that the source code for all the programs presented in this section can be found in appendix B.
3.2.1 Breadth-first graph search

We use the breadth-first graph search algorithm to show how the rationed-memory technique can decrease the amount of memory used by a queue. The algorithm uses a queue to keep track of the vertices reachable from the already visited nodes, and the order in which those reachable vertices should be visited. A classic implementation of the BF algorithm allocates a queue of size $n$ if the graph has $n$ nodes. Because it visits every node in the graph, the algorithm uses the entire space allocated for the queue, accessing every location in it.

Let us define the active portion of the queue as the interval that contains the nodes that have been detected as reachable, but they have not been visited yet. This active portion is the only part of the queue that is in fact required for the traversal, and therefore it is the only part which has to be kept in memory – the already visited nodes are not needed anymore, and there is no space required for the currently unreachable nodes.

When the breadth-first search implementation is compiled with the rationed-memory technique, the compiler allocates only a fraction of the required memory for the queue. As the search advances in the graph such that it moves beyond the allocated space in the queue, the rationed-memory compiler redirects the accesses to the queue to the hash for out-of-bounds writes. As long as the hash is large enough to fit the active portion, the breadth-first search completes successfully, as illustrated in figure 3-1. Therefore, even if the algorithm accesses every location in the size $n$ queue, in fact the compiler can reduce the memory consumption to only a fraction of that: the program will used only the allocated part of the queue, plus the space available in the hash.

The figure shows how breadth-first search progresses in a graph where the nodes form a grid. It can be seen that the maximum size of the queue active portion is approximately equal to the size of the largest level in the breadth-first search tree. If the nodes of the graph have small degrees on the average and the graph is not very dense, the branching factor in the breadth-first tree remains small and there are more
Figure 3-1: Breadth-first search with the rationed-memory compiling technique. The dotted lines between nodes represent edges in the graph. The thicker continuous line shows the order in which the algorithm traverses the nodes. The gray nodes represent the current active portion of the queue used in the algorithm, meaning that they are reachable, but not yet visited. The active portion has to fit in the hash for out-of-bounds writes in order for the search to complete successfully.
levels in the tree, which means that the largest level is smaller.

We tested our breadth-first search implementation compiled with the rationed-memory technique on a variety of input graphs. The program handled well a graph of 20000 nodes where the nodes were linked in a chain, even for a ratio of allocated versus requested memory as low as 0.5 and a small hash for out-of-bounds-writes, making for savings of 50% in memory consumption. We also ran the program on a grid graph such as the one in figure 3-1, but with 10000 nodes. The settings for which the breadth-first search was successful on this graph and it consumed the least memory were: the allocated vs. required memory ratio was 0.5 and the hash size was 1000. The reduction in memory usage in this case is 40%. On randomly generated graphs with up to 5000 nodes, we obtained successful runs with memory savings ranging from 25% to 40%.

3.2.2 Convex hull

We implemented the Graham scan algorithm for finding the convex hull of a set of 2D points. The algorithm starts by picking a point known to be a vertex of the convex hull, such as the rightmost lowest point in the set, and then radially sorts all the other points based on the angle they form with that initial point and the x-axis. After the sort, the algorithm loops through the set of points, testing whether they are vertices of the convex hull. This process uses a stack of points that holds the current segment of convex hull. During the loop, the algorithm pushes off the stack the points that no longer belong to the convex hull, and then pushes the current point onto the stack. At the end of the algorithm, the stack contains the entire convex hull.

We are interested in how much memory the stack needs. In the worst case, all the points in the set will be in the convex hull, and in that case the stack needs to hold all points. The conservative approach would be to allocate a stack of size $n$, if there are $n$ points in the set. However, in the average case much fewer points are needed: if the points are randomly distributed in a square, the average size of the convex hull is $O(\log n)$, while if the points are randomly distributed in a circular area, the average size of the convex hull is $O(n^{1/3})$ [5]. Even if during the selection process some points
<table>
<thead>
<tr>
<th>No. of points</th>
<th>Distribution</th>
<th>Maximum stack size</th>
<th>Convex hull size</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>square area</td>
<td>33</td>
<td>25</td>
</tr>
<tr>
<td>10000</td>
<td>square area</td>
<td>29</td>
<td>24</td>
</tr>
<tr>
<td>10000</td>
<td>square area</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>10000</td>
<td>circular area</td>
<td>45</td>
<td>42</td>
</tr>
<tr>
<td>10000</td>
<td>circular area</td>
<td>49</td>
<td>47</td>
</tr>
<tr>
<td>10000</td>
<td>circular area</td>
<td>46</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 3.1: The results for a sample of the tests performed on the convex hull program compiled with the rationed-memory compiler. For all the tests, the algorithm used only a small portion of the allocated stack and did not need to use the hash for out-of-bounds writes. The program ran successfully even when the ratio of allocated versus requested memory was 0.1.

that are not on the convex hull find their way onto the stack, the maximum size of the stack will not be significantly larger than the size of the convex hull.

Our test results support this intuition, showing that the stack is largely unused (see some sample tests and results in table 3.1) and that the rationed-memory technique achieves major memory savings in this case. We ran the convex hull program on randomly generated inputs, on sets of up to 10000 points, distributed either in a square area or in a circular area. We varied the amount of memory allocated to the stack, progressively lowering the ratio of allocated versus requested memory down to a minimum ratio of 0.1. For all tested ratios and for all our input sets of points, the program ran successfully and produced correct results. Because it never attempted to access a stack location past the allocation limit, the program did not use the hash for out-of-bounds writes, making for total memory savings of about 90% for the stack, and about 45% for the entire program.

### 3.2.3 Single source shortest paths

We implemented the Dijkstra algorithm for finding single source shortest paths in a graph, and we focused our experiments on the memory usage of the path vectors, as another example of a data structure prone to over-allocation, and for which the rationed-memory compiler could improve the memory usage.

The Dijkstra algorithm generates a tree in which the root is the given source node,
and the paths from the root to any other node are shortest paths in the initial graph. The length of a path can be anywhere from 1 to \( n \), if there are \( n \) nodes in the input graph, so a conservative implementation would allocate a memory block of size \( n \) to hold the path. If we are to generate all shortest paths from a single source node, the total allocated space would be proportional to \( n^2 \), when in fact the space actually used is much smaller: in the worst case, the shortest paths tree looks like a chain, and the space used if we generate the paths is roughly \( n^2/2 \). For average cases, the memory usage drops dramatically.

We tested our implementation of the Dijkstra algorithm compiled with the rationed-memory compiler on randomly generated graphs with up to 200 nodes, and on all tests the amount of used memory for keeping the shortest paths was well under what the compiler allocated, even when we used a 0.5 ratio of allocated versus requested memory. We also tested our program on graphs with up to 200 nodes that produced the described worst-case scenario, and the runs were successful, for an allocated/requested ratio as small as 0.5 and a hash for out-of-bounds writes with 500 entries. The memory savings in this case were close to 50%.

### 3.2.4 Eratosthenes’ prime number sieve

One of the programs that we tested was Erathosthenes’ sieve for generating prime numbers. The program exhibited interesting behavior when we pushed down the size of the hash for out-of-bounds accesses and the fraction of memory allocated on a `malloc` call compared to the requested size, such that the data needed by the sieve could not fit all into the available memory at the same time.

The prime number sieve uses an array of flags, one flag per number, and each flag signals whether its number is prime. When the fraction of allocated versus requested memory and the size of the hash were large enough for the entire array of flags to fit into memory, or in other words, for the program to be able to access every flag in the array, the sieve implementation worked fine, as expected. However, more interesting was the case when the array could not fit into the available memory, and entries in the array, i.e. flag values, were thrown out of the hash.
Table 3.2: Test results for Erathostene’s prime number sieve compiled with the rationed-memory compiler. In all tests, the program had to determine all the prime numbers smaller than 40000 (there are 4203 such numbers). We varied the compiler parameters, decreasing the amount of memory made available until the program failed to complete its execution. For the successful runs, we counted the correctly determined primes. For each of the runs, the output did not contain any composite numbers. The execution status column specifies whether entries had to be thrown out of the hash for out-of-bounds accesses, which means that the available memory was less than the 40000 bytes needed by the algorithm.

We ran the sieve program to determine all prime numbers smaller than 40000, and we varied the rationed-memory compiler parameters: the hash size and the amount of memory returned on a malloc call. Table 3.2 shows the results of our tests. It is worth noting that the program did not output any non-prime numbers for any of the runs, even when the compiler provided less memory than the sieve algorithm needs.

### 3.3 Discussion

In this chapter, we displayed a diverse set of instances of over-allocation, both in a large application, such as Pine, and in a series of commonly encountered algorithms.
The large range of ways in which programs can over-allocate memory for their data structures shows that over-allocation is an ubiquitous problem. While for some of the shown instances there may be particular management techniques or tricks that could alleviate the problem, we claim that the rationed-memory technique is well-suited to tackle all instances of over-allocation.

Our experiments on several small programs compiled with the current implementation of the rationed-memory compiler confirmed our claim, showing that the technique produces savings of 25 to 50% on several types of data structures. Moreover, the data structures tested are not used only by small programs, but in fact they are commonly used in all kinds of applications. For example, the technique could replicate the memory savings produced for the queue in the breadth-first search algorithm in queues used in other algorithms.

While it is hard at this point to gauge the memory savings that the rationed-memory technique could produce in larger programs, our results indicate good promise for the effectiveness of the technique in the context of more complex systems. The next chapter discusses some of the steps needed in order to apply the technique to such systems.
Chapter 4

Future work

The directions of future work presented in this chapter belong to two different categories, depending on their impact on the functionality and performance of the compiler. The first section of this chapter presents the first category, which comprises the upgrades needed for compiling larger applications with the rationed-memory compiler. The second section examines enhancements that could improve the performance of the technique.

4.1 Compiler functionality upgrades

The previous chapter presented the results of tests performed on small programs compiled with the rationed-memory compiler. The next stage of this project would be to compile larger applications, such as open-source C programs, and observe their behavior in the rationed-memory environment. Currently, the applications that we compiled with our compiler do not run successfully because some of the cases of memory access are not handled correctly in the rationed-memory context.

One such case is when a program accesses the fields of a variable of a struct type. A program usually requests a memory block with malloc to hold such variables. The rationed-memory compiler makes no distinction between malloc calls for arrays and those for struct variables, so it resolves the malloc call in the manner described in chapter 2: it allocates a block smaller than the requested size, and it returns the
address of that block. However, the compiler treats differently the reads or writes to fields of the \textbf{struct} variable than reads and writes to elements of a dynamically allocated array. We have explained before that for arrays, the compiler redirects the accesses past the allocated memory block to the hash for out-of-bounds accesses. For \textbf{struct} variables, the accesses past the allocated memory zone are not handled in the same way; instead, the program is allowed to access the actual memory location at which the accessed field would be located if the memory allocation would take place in the traditional fashion, not using the rationed-memory compiler. Unfortunately, this leads to memory corruption and eventually to crashes through segmentation fault. This manner of handling accesses is inherited from the version of the CRED compiler augmented with boundless memory blocks on top of which we implemented the rationed-memory compiler.

To fix the described problem, the compiler has to accomplish two things. The first change is to allow for the accesses to fields situated past the allocated memory zone for a \textbf{struct} variable to go to the hash for out-of-bounds accesses. The second, more subtle point, is that when allocating memory for a \textbf{struct} variable, the compiler should make sure that the end of the allocated block falls at the boundary between fields in the variable. In other words, there should be no fields that span across the end of the allocated memory zone. If it were the case that such a field existed, the compiler would probably consider it an in-bounds field and allow the program to access the memory location at which the field starts. However, a write in that field may go past the allocated memory zone and it may cause memory corruption. Therefore, it is essential that the compiler aligns the boundary of the allocated memory zone with a boundary between consecutive fields in the \textbf{struct} variable.

Another modification to the compiler is applying the rationing technique to the statically allocated memory. Currently, the compiler does not intercept the static allocation of arrays, which implies that it cannot reduce the size of the allocated block to achieve savings in memory usage. Obviously, legacy C programs use statically allocated arrays, so to provide the full advantage of the technique to those programs without intervening in their source code, the compiler should intercept those allocat-
tions and proceed similarly to the handling of malloc calls, as described in chapter 2.

4.2 Ideas for enhancement

This section presents several ideas that would contribute to the effectiveness of the rationed-memory technique, although they are not required by the compiler in order to be fully compatible with applications written in legacy C code.

Profiling the memory usage of programs

The performance of the rationed-memory technique when applied to a program is contingent on how the two main settings of the compiler (the ratio of allocated versus requested memory, and the size of the hash for out-of-bounds accesses) impact the program. That impact depends on how the program uses the memory it allocates. For example, if the degree of over-allocation in the program is very high, meaning that the program will not use much of the memory it requested, then likely the best settings for the compiler in that case would be a low ratio of allocated versus requested memory, and a hash large enough to fit all out-of-bounds accesses. If, on the other hand, the program does a good job of estimating how much memory it needs, meaning that the degree of over-allocation is low, then the compiler should be set for a high ratio of allocated vs. requested memory; if we are trying to save more memory by pushing that ratio down, then the size of the hash will have to go up in order for the hash to fit more out-of-bounds writes, and the savings in memory usage will in fact decrease. Further, redirecting memory accesses to the hash incurs an overhead cost on the program and decreases its performance.

Therefore, careful selection of the compiler settings for a specific program could make a big impact on the performance of the technique in that case. One way to produce a good selection of settings would be to perform trial runs of the program with different parameters, measure the performance for each case, and then use the best parameters for a final executable.
Dynamically adjusting the allocated memory

This idea also implies adjusting the compiler parameters, but it involves doing so during the execution of the program. For example, the compiler would have a default ratio of allocated versus requested memory, and it would initially perform all the memory allocations using that ratio. However, if during the execution, for some particular data structure, the program accessed many locations past the boundary of the allocated memory, the compiler would be able to detect that situation and perform a reallocation, increasing the amount of memory allocated for that data structure. This adjustment would eliminate the redirection of some reads and writes to the hash; the accesses would take place with less overhead, and the memory previously used by the hash for those reads and writes would be freed up.

Similarly, if the compiler would detect that the program is not using all the allocated memory for a data structure, it could perform a reallocation to reduce the allocated memory for that data structure. That modification would free up memory, while the hash for out-of-bounds accesses would be able to handle the few accesses past the new boundary of allocated memory.

Both of these modifications should avoid placing a high overhead on the performance of the program; the compiler should perform the run-time checks and the possible consequent reallocations rarely enough that the performance of the program does not decrease significantly.

Decreasing the overhead of auxiliary data structures in the compiler

In chapter 2 we described some additional data structures used by the bounds-checking compiler and by the boundless memory blocks compiler. Those data structures — the object table, out-of-bounds objects and the hash for out-of-bounds writes — are present in the rationed-memory compiler as well. While those data structures are essential to the functioning of the rationed-memory compiler, the memory that they use counts against the performance of programs compiled with our technique.
Therefore, any reduction in the memory usage of these data structures would add to the total savings produced by the rationed-memory technique.
Chapter 5

Related work

In this chapter we discuss a series of techniques related to our work in the area of reducing the memory usage of programs. While some of these techniques tackle different facets of the problems of over-allocation and memory waste in programs, their purpose is the same as the purpose of the rationed-memory technique, namely reducing the footprint of a program in memory.

5.1 Data size optimizations for object-oriented programs

Our technique primarily targets arrays and buffers, dynamically or statically allocated, and the previous chapters discussed our technique and offered examples mainly in the context of procedural programming. The technique would likely prove as effective for arrays in the context of object-oriented programming, but it does not account for other manners in which object-oriented programs waste memory: programmer-defined fields with constant values or with a limited range of values, unread fields, fields with common default values or usage patterns. Ananian and Rinard in [2] identified this set of problems and developed a series of analysis algorithms to automatically detect them, as well as corresponding optimization techniques to reduce the memory consumption of programs in these cases. Tests performed using an im-
plementation of the analysis algorithms and optimizations for Java programs showed a reduction of up to 40% in the maximum live heap size required for the programs.

5.2 Eliminating equivalent object instances and dead data members

Marinov and O’Callahan in [7] present a technique which performs a dynamic analysis to detect equivalent object instances. Equivalence classes of such instances can be replaced with a single representative object. On two benchmarks, the technique produced a reduction of the space used by 47% and 38%, respectively.

Sweeney and Tip in [13] also look at object-oriented programs, but they study dead data members in C++ applications. They define dead data members as those data members whose removal from the application would not affect its functionality. Dead data members can occur as a result of unused functionality in class libraries, of differences between the initial design of a class and its actual use in applications, or because of increased complexity and growing class hierarchies in applications. The study presents an algorithm that performs a whole-program analysis on an application and identifies dead data members. The results of tests performed on various C++ benchmark programs show that on average 12.5% of the data members in the benchmarks are dead, and 4.4% of the space used by those programs is taken by dead data members.

5.3 Detecting and exploiting narrow bitwidth computations

While our technique takes advantage of unused portions of arrays and data structures to reduce the memory consumption of programs, another line of research looks to obtain results by performing bitwidth analysis and taking advantage of unused bits [2, 3, 10, 12]. One of these studies presents a tool which can detect unused
or constant bits in C programs using dataflow analysis, and shows evidence that in some programs, up to 31% of the computed bytes are useless in the execution of the programs and get thrown away [3].

Zhang and Gupta in [14] proposed two transformations that aim to compress the representation of dynamic data structures used in programs. The common-prefix transformation compresses a 32 bit address pointer to a 15 bit entity, using the locality properties of pointers in the program, while the narrow-data transformation compresses a 32 bit integer into a 15 bit entity, if the possible values of the integer fit into a 15-bit range. Pairs of such 15 bit entities can be packed in a 32 bit word. Tests have shown average reductions of 25% in heap allocated storage.
Chapter 6

Conclusion

We presented *rationed-memory compiling*, an useful technique for reducing the memory consumption of programs. This thesis describes the inner workings of our implementation of a rationed-memory C compiler, compatible with programs written in standard C language. We tested a series of programs compiled with our compiler and we showed that the technique can produce important memory savings for various types of data structures. We exhibited cases of over-allocation in a widely used open source program like Pine, demonstrating the potential of the rationed-memory technique for larger applications. We also offered several ideas for refining the technique, and we reviewed related academic studies on the topic of memory consumption reduction.
Appendix A

Over-allocation in Pine 4.44 – source code

The first example of over-allocation in Pine presented in section 3.1 describes how Pine creates a full-form To field expanded from the user input in the message editing window. This operation is performed by the `build.address.internal` function in the source file `bladdr.c`, lines 3944-4259. Here is the header of the function, as well as the comments that precede it:

```c
/*
 * Given an address, expand it based on address books, local domain, etc.
 * This will open addrbooks if needed before checking (actually one of
 * its children will open them).
 * *
 * Args: to -- The given address to expand (see the description
 * in expand_address)
 * in expand_address
 *   full_to -- Returned value after parsing to.
 *   error -- This gets pointed at error message, if any
 *   fcc -- Returned value of fcc for first addr in to
 *   no_repo -- Returned value, set to 1 if the fcc or lcc we're
 * returning is not reproducible from the expanded
 * address. That is, if we were to run
 * build_address_internal again on the resulting full_to,
 * we wouldn't get back the fcc again. For example,
 * if we expand a list and use the list fcc from the
 * addrbook, the full_to no longer contains the
 * information that this was originally list foo.
 * save_and_restore -- restore addrbook state when done
*/
```
* Result: 0 is returned if address was OK,
* -1 if address wasn’t OK.
* The address is expanded, fully-qualified, and personal name added.
* Input may have more than one address separated by commas.
* Side effect: Can flush addrbook entry cache entries so they need to be
  * re-fetched afterwords.
*/

int build_address_internal(to, fullto, error, fcc, no_repo, lcc,
    save_and_restore, simple_verify, mangled)

  BuildTo to;
  char **full_to,
  **error,
  **fcc,
  **lcc;
  int *no_repo;
  int save_and_restore, simple_verify;
  int *mangled;

The over-allocation instance occurs on line 4051 of the file bldaddr.c, when Pine
allocates a buffer of size about MAX_ADDR_FIELD, which is defined as 10000 in the os.h
header file. The fs.get function below calls malloc to get a memory block of the
specified size:

tmp = (char *)fs_get((size_t)(MAX_ADDR_FIELD + 3));

The second example of over-allocation presented in section 3.1 deals with parsing
the From fields of messages in the INBOX folder. Pine requests a string buffer in which
to copy the contents of that field in function addr.list.string, in file bldaddr.c,
lines 7089-7123. The allocation call is:

list = (char *)fs_get((size_t)est_size(adrlist));

The size of the requested block is computed by the est.size function which is
found in file bldaddr.c, lines 7257-7285. The function computes a maximum length
for the buffer as a function of the address specified in the e-mail message, taking into
account that special characters are expanded into 2-character escape sequences.
/ * Compute an upper bound on the size of the array required by
  * rfc822_write_address for this list of addresses.
  *
  * Args: adrlist -- The address list.
  *
  * Returns -- an integer giving the upper bound
  */
  int
  est_size(a)
    ADDRESS *a;
{
    int cnt = 0;

    for(; a; a = a->next){

      /* two times personal for possible quoting */
      cnt += 2 * (a->personal ? strlen(a->personal) : 0);
      cnt += (a->mailbox ? strlen(a->mailbox) : 0);
      cnt += (a->adl ? strlen(a->adl) : 0);
      cnt += (a->host ? strlen(a->host) : 0);

      /* add room for:
       *   possible single space between fullname and addr
       *   left and right brackets
       *   @ sign
       *   possible : for route addr
       *   , <space>
       *
       * So I really think that adding 7 is enough. Instead, I’ll add 10.
       */
      cnt += 10;
    }

    return(max(cnt, 50)); /* just making sure */
}
Appendix B

Programs used in experiments

B.1 Breadth-first search

Here is the source code for the breadth-first search discussed in section 3.2:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_NEIGHBORS 10

int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Usage: %s <graph_input_file> <output_file> \n", argv[0]);
        exit(1);
    }

    FILE *inFile;
    inFile = fopen(argv[1], "r");

    int n;

    // read the graph from the input file in the form of adjacency lists
    fscanf(inFile, "%d\n", &n);

    int adj[n][MAX_NEIGHBORS], nadj[n];

    int i,j,k,xx;
    for (i = 0; i < n; i++) {
        fscanf(inFile, "%d", &xx);
```
nadj[i] = xx;
for (j = 0 ; j < nadj[i] ; j++) {
    fscanf(inFile, "%d", &xx);
    adj[i][j] = xx;
}
fclose(inFile);

FILE *outFile;
outFile = fopen(argv[2], "w");

// prepare auxiliary data structures for the breadth-first search
char v[n];
int* q = malloc(n * (sizeof(int)));

for (i = 0 ; i < n ; v[i] = 0, i++);
q[0] = 0; v[0] = 1;
fprintf(outFile, "0\n");
int begin = 0, end = 0, nvisited = 1;

while ((nvisited < n) && (begin <= end)) {
    k = q[begin]; xx = 0;
    for (i = 0 ; i < nadj[k] ; i++) {
        j = adj[k][i];
        if (v[j] == ) {
            end++; q[end] = j;
            fprintf(outFile, "%d\n", j);
            v[j] = 1;
            nvisited++; xx = 1;
        }
    }
    begin++;
    if ((xx == 0) && (begin > end)) {
        for (i = 0; (i < n) && (v[i]) ; i++) ;
        end++; q[end] = i;
        fprintf(outFile, "%d\n", i);
        v[i] = 1;
        nvisited++;
    }
}
fclose(outFile);
B.2 Convex hull

We show here the source code for the convex hull implementation we used for the tests described in section 3.2. Note that we only display the part of code that performs the selection of the convex hull vertices, as that is the part of code relevant to our tests.

/******
 * convexhull.c
 * Computes the convex hull of a given set of points.
 * */
#include <stdio.h>
#include <stdlib.h>

struct point {
    long x, y;
};

struct point *p, *h, miny;
int n, nh;

long det_coords(long, long, long, long, long, long);

int main(int argc, char **argv) {

    if (argc != 3) {
        printf("Usage: %s <infile> <outfile>
\n", argv[0]);
        exit(1);
    }

    FILE *inFile;
    inFile = fopen(argv[1], "r");

    fscanf(inFile, "%d\n", &n);

    if ((h = (struct point *) malloc (n * sizeof(struct point))) == NULL) {
        printf("Error, memory not allocated.\n");
        exit(1);
    }

    // read the points from the input file and perform the selection
    // points are given in sorted order, as we are interested only in the
    // convex hull vertex selection process
    int i, xx, yy;
nh = 3;
scanf(inFile, "%d %d\n", &xx, &yy);
h[0].x = xx; h[0].y = yy;
scanf(inFile, "%d %d\n", &xx, &yy);
h[1].x = xx; h[1].y = yy;
scanf(inFile, "%d %d\n", &xx, &yy);
h[2].x = xx; h[2].y = yy;

for (i = 3 ; i < n ; i++) {
    scanf(inFile, "%d %d\n", &xx, &yy);
    for ( ; det_coords(h[nh-2].x, h[nh-2].y, 
        h[nh-1].x, h[nh-1].y, 
        xx, yy) >= 0 ; nh -- ) {}
    h[nh].x = xx; h[nh].y = yy;
    nh++;
}

// write the convex hull points into the output file
FILE * outputFile;

outFile = fopen (argv[2], "w");
fprintf(outFile,"%d\n", nh);
for (i = 0 ; i < nh ; i++) {
    fprintf(outFile, "%d %d\n", h[i].x, h[i].y);
}
fclose(outFile);

long det_coorts (long x1, long y1, 
    long x2, long y2, 
    long x3, long y3) {
    return (y2 - y3) * (x1 - x3) + (x3 - x2) * (y1 - y3); 
}
B.3 Single source shortest paths

Here is the implementation of the single source shortest path algorithm used in section 3.2:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_COST 100

int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Usage: %s <graphinput_file> <outputfile> \n", argv[0]);
        exit(1);
    }
    FILE *inFile;
inFile = fopen(argv[1], "r");
    int n, m;

    // read the graph from the input file as a list of edges
    fscanf(inFile, "%d\n", &n);
    fscanf(inFile, "%d\n", &m);
    int edges[m][3];
    int i, j, k, xx, yy, cc, min;
    for (i = 0; i < m; i++) {
        fscanf(inFile, "%d %d %d\n", &xx, &yy, &cc);
        edges[i][0] = xx;
        edges[i][1] = yy;
        edges[i][2] = cc;
    }

    int c[n], t[n];
    char v[n];
    for (i = 0; i < n; c[i]=n*MAX_COST+1, v[i] = 0, t[i] = -1, i++) ;
    c[0] = 0;

    k = 0;
    while (k < n) {
        min = n*MAX_COST+1; j = -1;
        for (i = 0; i < n; i++) {
            if (c[i] < min) {
                min = c[i]; j = i;
            }
        }
        c[j] = MAX_COST;
        t[j] = k;
        v[j] = 1;
        for (i = 0; i < m; i++) {
            if (edges[i][0] == j) {
                int cc = edges[i][2];
                int nx = edges[i][1];
                if (c[nx] > c[j] + cc) {
                    c[nx] = c[j] + cc;
                    t[nx] = k;
                }```
```
if ((v[i] == 0) &&
    (c[i] < min)) {
    min = c[i]; j = i;
}
}

v[j] = 1; k++;
for (i = 0 ; i < m ; i++) {
    if ((edges[i][0] == j) &&
        (edges[i][2] + c[j] < c[edges[i][1]])) {
        c[edges[i][1]] = c[j] + edges[i][2];
        t[edges[i][1]] = j;
    }
    if ((edges[i][1] == j) &&
        (edges[i][2] + c[j] < c[edges[i][0]])) {
        c[edges[i][0]] = c[j] + edges[i][2];
        t[edges[i][0]] = j;
    }
}

FILE * outFile;
outFile = fopen(argv[2], "w");

int *p;
for (k = 0 ; k < n ; k++) {
    p = malloc (n * sizeof(int));
    i = k; j = 0;
    while (i > -1) {
        p[j] = i;
        i = t[i];
        j++;
    }
    for (i = j-1 ; i > -1 ; i--) {
        fprintf(outFile, "%d ", p[i]);
    }
    fprintf(outFile, "\n");
}
fclose(outFile);
B.4 Eratosthenes’ prime number sieve

Here is the source code of our implementation of Erathostene’s prime number sieve, whose behavior under rationed-memory compiling is discussed in section 3.2:

```c
/*****
* sieve.c
*
* Erathosthenes’s sieve.
*
*/
#include <stdio.h>
#include <stdlib.h>

int max;
char *s;

int main(int argc, char **argv) {

    if (argc != 3) {
        printf("Usage: %s max_number <outfile>\n", argv[0]);
        exit(1);
    }

    max = atoi (argv[1]);

    if ((s = (char *) malloc (max * sizeof(char))) == NULL) {
        printf("Error, memory not allocated.\n");
        exit(1);
    }

    int i,j;
    for (i = 2 ; i < max ; i++) {
        s[i] = 1;
    }

    FILE *outFile;
    outFile = fopen(argv[2], "w");

    for (i = 2 ; i < max ; i++) {
        if (s[i] == 1) {
            fprintf(outFile,"%d\n",i);
            for (j = i*i ; j < max ; j+=i) {
                s[j] = 0;
            }
        }
    }
```
fclose(outFile);
}
Bibliography


