UNIVERSITY OF WATERLOO
Faculty of Mathematics

DESIGN DECISIONS FOR EFFICIENT EXTENSIBLE MARKUP LANGUAGE PARSING

Crank Software Inc.
Storyboard Software Developer (co-op)
Kanata, Ontario, Canada

Samuel Wanuch
20521149
2B Computer Science
January 14, 2016

Table of Contents

List of Figures

Summary

The purpose of this report is both to justify the decisions made in the design of a memory efficient Extensible Markup Language (XML) parser, and to give insight to any employees who may have to work with the XML parser into how it works, and ways it can be improved.

The analysis will cover the both the differences in performance and reasons for those differences between versions with and without the following features:

- Reference counting strings
- String handles
- Implicit XML structuring
- Unique attribute name sets
- Batch allocation of attribute value arrays
- Batch allocation of XML nodes

The final result of these optimizations is an XML parser which is at worst 3 times slower, but uses almost 10 times less memory compared to the existing XML parser.

Further improvements in speed can be made by investigating more efficient ways to compare a string to the existing reference counted strings. To improve memory, it is possible that certain XML tags in the file format will always have the same tag name when they have the same attribute names, which would allow only unique combinations to be stored, instead of storing tag names individually.

1.0 Introduction

The main product of Crank Software Inc. is a user interface solution which allows for quick prototyping and and quality user interfaces across a wide range of devices (Crank Software Inc.: About). This range of devices stretches from high end computers to embedded systems with extremely limited resources. On the embedded systems, starting up the user interface with the Storyboard Engine is a difficult task, as multiple parts of the engine are allocating large amounts of memory. In order to improve the performance of the Storyboard Engine, especially on these constrained systems, it was decided that the as of yet unoptimized Extensible Markup Language (XML) parser which reads in the description of the user interface should be rewritten to be more efficient.

The goal of the efficient XML parser is to swiftly parse read only XML files into an easy to use representation with the minimum amount of memory allocations. While speed is important, minimizing memory allocations is more valuable.

1.1 Extensible Markup Language

The Extensible Markup Language, otherwise known as XML, is a flexible file format designed to be both human and machine readable, emphasizing simplicity, generality, and usability (Extensible Markup Language: Introduction). It is based on using nested named tags which have both attributes, and contents. Here is an example:

Listing 1: XML Example

```xml
<tag>
        <tag-name attribute-name="attribute-value">
                Contents of tag-name
                <nested-tag a="1" b="2">
                        Contents of nested-tag
                </nested-tag>
                <other-nested-tag>
                </other-nested-tag>
        </tag-name>
</tag>
```

The sample above has 4 tags, with the names ”tag”, ”tag-name”, ”nested-tag”, ”other-nested-tag”.

The tag named ”tag-name” has the attribute ”attribute-name” with a value of ”attribute-value”. The tag named ”nested-tag” has the attributes ”a” and ”b”, with the values of ”1”, and ”2” respectively.

The tag named ”tag-name” has the text contents ”Contents of tag-name”, and the tag named ”nested-tag” has the contents ”Contents of nested-tag”.

The tag named ”tag” has one child, the tag named ”tag-name”. That tag in turn has two children, the tags named ”nested-tag” and ”other-nested-tag”, neither of which have children.

1.2 ANSI C89

In order to ensure the XML parser is compatible with as many systems as possible, it will be written in the C standard created by the American National Standards Institute (ANSI), specifically the standard referred to as C89.

ANSI C89 guarantees the following maximum value for these basic data types (ANSI C89):

unsigned char	255
unsigned short	$65,535$
unsigned int	$65,535$

In order to encode this range, that means the data types must at minimum have the following number of bytes:

unsigned char	1 byte
unsigned short	2 bytes
unsigned int	2 bytes

Of particular interest is the fact that a short may be the same size as an int on some platforms. Since using an int and expecting it to be the more expected 4 bytes long would not be portable, and expecting it to be the same size is effectively the same as using a short, any data type which would have been an int in the parser will instead be a short. No larger basic data types are used apart from pointers.

All tests for the original XML parser and the various new versions were done on a 64-bit virtual machine running Ubuntu 15.10, and compiled using GCC with the flag -O3 for optimization. The most important result of this is that pointers in the program are actually 8 bytes long.

1.3 Memory Allocation

The reader is expected to be familiar with the concept of allocating memory on the heap. However, the way this memory is allocated is not necessarily common knowledge.

All heap memory used in the implementations will be allocated via malloc. While ideally malloc would always return exactly the requested amount of space, and never prevent any other memory from being used, the reality of the problem is otherwise.

The ANSI C89 standard states that the memory returned by malloc ”is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated” (ANSI C89). This means that malloc may align all allocated memory to a certain byte boundary, such as 4, 8, or 16 bytes. While this may seem insignificant, it does mean that allocating an object with a size that isn’t a multiple of 4 or 8 or 16 wastes a certain amount of bytes. For instance, if the memory is aligned to a 4 byte boundary, then allocating an object which is 6 bytes large wastes 2 bytes, as the 2 bytes immediately after the object cannot be allocated for another object as they do not begin at the 4 byte boundary.

Additionally, as per Valgrind's massif, there are two sources of "wasted" memory that was not asked for (Valgrind Massif: User Manual):

Administrative Data Each block of memory returned by malloc has a number of additional bytes of data immediately before it. Generally this is used by malloc to record the size of the allocated block.

Rounding Many implementations of malloc round up the number of bytes requested to a larger number, this is essentially the same as the alignment issue mentioned above.

Another source of wasted memory is fragmentation. This problem arises when memory is continually allocated and freed, and the available memory becomes fragmented into small scattered blocks. This can result in a large amount of memory being available, but unable to be used as it is in such small pieces surrounded by other allocated blocks that no free piece is large enough to use.

1.4 Valgrind Massif

In order to measure the speed and memory usage of the various implementations, the Valgrind massif tool will be used.

Valgrind is a well known framework for dynamic analysis tools which can analyze running programs. It is most well known for it's default tool Memcheck, which records memory allocations and frees to detect a wide range of memory related errors (Valgrind Memcheck: User Manual).

The Massif tool focuses on analyzing when and how much memory is allocated by a program (Valgrind Massif: User Manual). The memory is measured in bytes, although for larger amounts, 1 kilobyte (kb) will refer to $1,000$ bytes, 1 megabyte (mb) will refer to $1,000$ kilobytes. In order to give a reliable measure of time while the program runs, massif measures the number of machine instructions a program uses before it completes. This is likewise referred to in terms of $1,000$ instructions (i) per kilo-instruction (ki), $1,000$ kilo-instructions per mega-instruction (mi), and $1,000$ mega-instructions per giga-instruction (gi).

Massif differentiates between the requested memory allocated, and the extra memory allocated due to the issues mentioned in the Memory Allocation section above. Performance comparisons will be based off of the total memory allocated, which is the sum of the two.

All performance tests will be using the following command: "valgrind –tool=massif –stacks=yes –max-snapshots=1000 –detailed-freq=1". The memory an implementation uses on a given test will be the memory used as reported by massif at peak memory usage.

1.5 Analysis Method

All analysis will be done using the Valgrind massif tool. the graphs shown will be from the massif-visualizer tool (Massif Visualizer: Overview). Each implementation of the XML parser will be run on three distinct inputs, representing the smallest valid input, an average use case input, and the largest existing input which could be found. These three test cases will be referred to respectively as small, medium, and big.

It should be noted that test case is more important than the small or large test cases. The small test case is tiny enough that any reasonable use case is at a minimum, twice as large. The big test case is massive enough that even the next largest use case is one fourth the size. The small and big test cases are meant to represent the lower and upper limits of size the XML parser is expected to be able to handle.

In order to justify a design choice, the final implementation's performance will be compared to the performance of a version which differs only in that it makes the opposite choice. Performance will be compared primarily based on the memory used by each version, and to a lesser extent, based on the speed of each version, on the three test cases. All numbers will be recorded to 4 significant digits.

1.6 Design Choices

The following design choices will be examined in this report:

- Reference counting strings
- Using string handles
- Implicitly defining the XML structure
- Storing only unique attribute name sets
- Batch allocation of attribute value arrays
- Batch allocation of XML nodes

2.0 Analysis

2.1 Final Implementation

The final implementation is the version that all other versions will be compared to. When parsing the empty file, this implementation uses 5.664 kilobytes.

The small test case is the minimum viable file, with a file size of 1,206 bytes, and the final implementation parses it using 14.30 kilobytes of memory, completing in 435.6 thousand instructions. This test case wastes 2,166 bytes of memory.

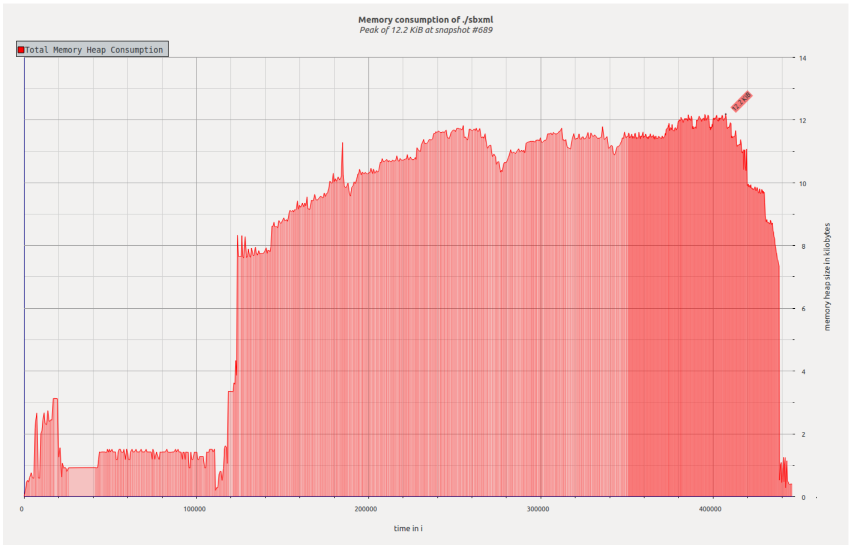

Figure 1: The final implementation running on the small test case: 14.30 kb in 435.6 ki

The medium test case is part of a user interface created by a co-worker, and a good indication of an expected use case, with a file size of 182,754 bytes. The final implementation parses it using 133.4 kilobytes of memory, completing in 50.27 million instructions. This test case wastes 21.30 kilobytes of memory.

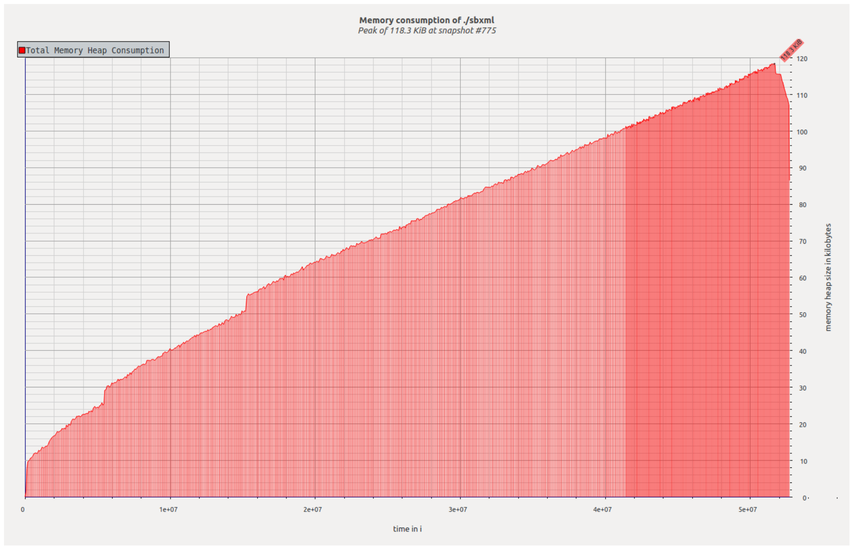

Figure 2: The final implementation running on the medium test case: 139.1 kb in 50.27 mi

The large test case is the largest use case that could be found, and represents the upper limit of size the parser will be used on, it is 796, 995 bytes large. The final implementation parses it using 533.8 kilobytes of memory, completing in 250.8 million instructions. This test case wastes 75.48 kilobytes of memory.

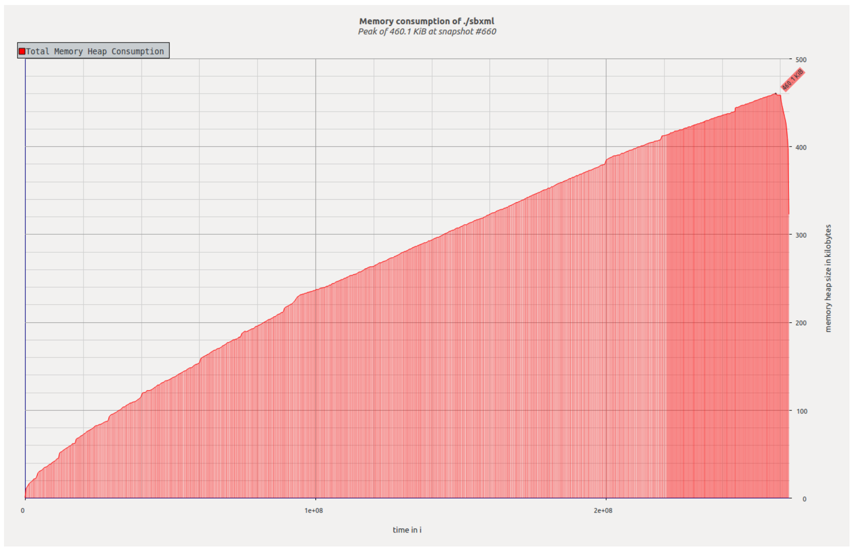

Figure 3: The final implementation running on the big test case: 533.8 kb in 250.8 mi

It is worth nothing that in each case except the small test case, the final implementation parses and represents the input file using less memory than the size of the file.

2.2 Reference Counting Strings

Reference counting strings proved to be one the most effective ways of limiting memory usage for the XML parser, as well as one of the largest impacts on speed. This was implemented by reading in strings from the file one at a time, checking each time to see if another equal string already existed; referencing such a string when it did exist and starting to track any new string.

The following graphs compare the final implementation on the left to a version that does not reference count strings on the right:

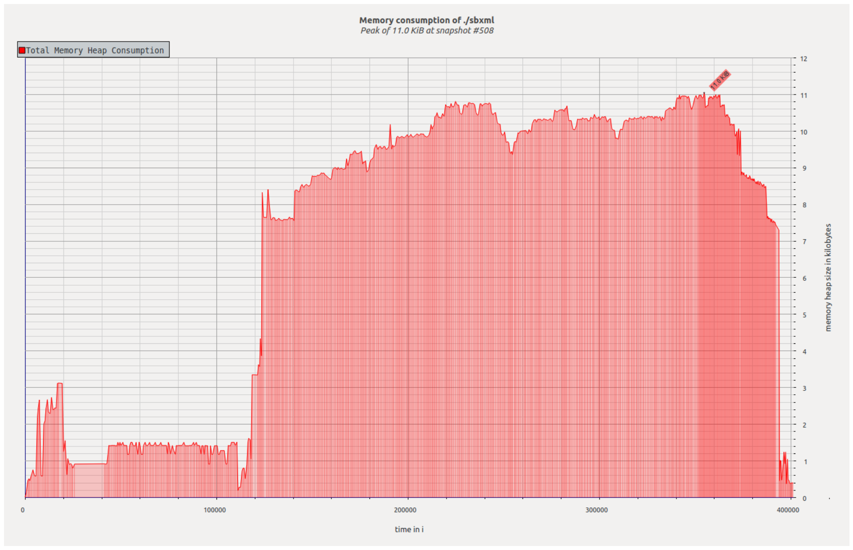

Figure 4: Parsing the small test case without reference counting strings: 13.15 kb in 391.9 thousand instructions

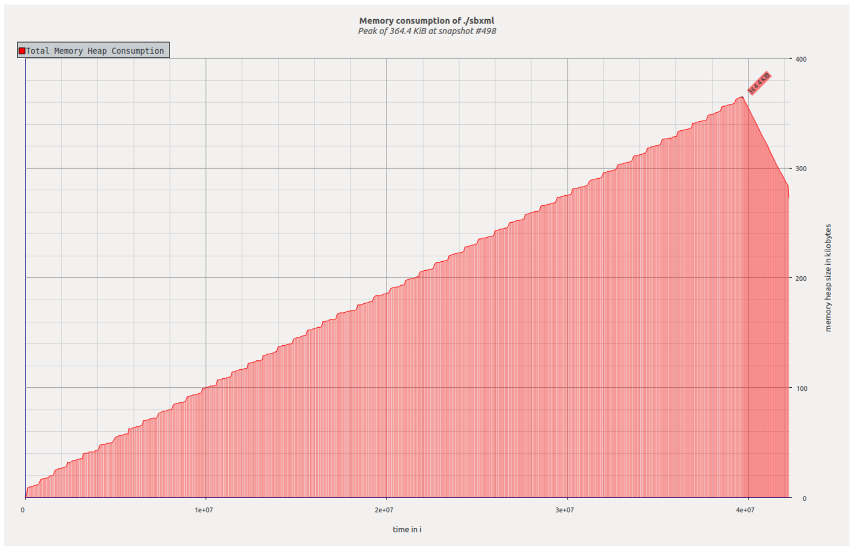

Figure 5: Parsing the medium test case without reference counting strings: 591.6 kb in 40.31 million instructions

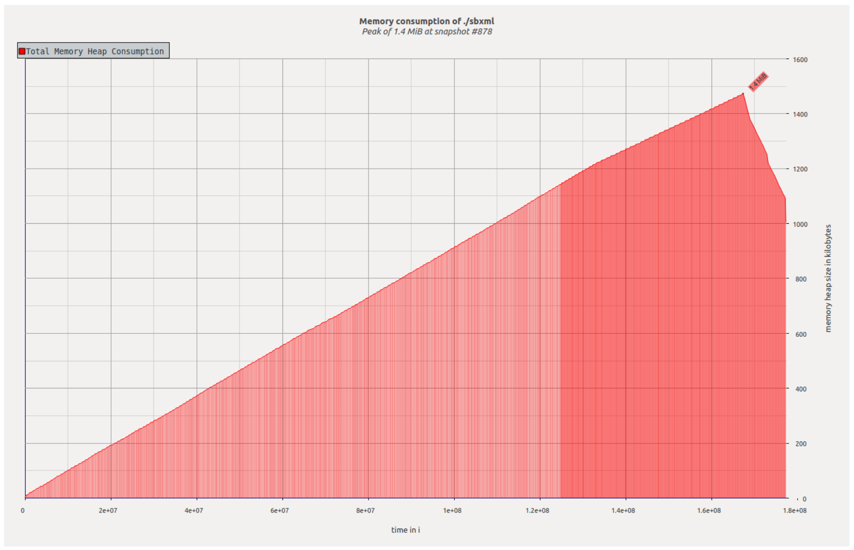

Figure 6: Parsing the big test case without reference counting strings: $2,334$ kb in 169.2 million instructions

As the graphs show, the final implementation uses significantly less memory than the non-reference counted version in the medium and big case, but slightly more in the small case. In the small case this can be put down to there being so few strings that reference counting has little benefit, along with the memory required for extra data structures used for efficient reference counting. In the medium and big case, the final implementation uses 24% and 23% of the memory respectively.

Thanks to the nature of the file format, many strings are repeated, such as tag names and attribute names, and often many attribute values are repeated as well. Reference counting also benefits from standardized naming system, as these are likely to produce identical strings. These factors allow the final implementation to use about one fifth of the memory of a non-reference counting implementation.

The runtime costs of reference counting can be seen here as well, as the final implementation is consistently slower, running 1.11, 1.25, and 1.48 times slower, compared to the non-reference counting version. This is due to reference counting requiring each new string to be compared to be compared to potentially equal strings, the cost of which only increases as more unique strings are added.

Overall, using reference counted strings slows down performance anywhere from 10% to 50%, but makes up for it by decreasing memory usage to almost as low as 20%. Therefore, reference counting strings is the efficient choice.

2.3 String Handles

The majority of information used to represent the XML file being parsed is strings. The simplest way to reference these strings is just to store a pointer to a string anywhere a string is needed. Since strings need to be referenced in so many places, they should be referenced as efficiently as possible. While pointers are the obvious data type, an unsigned short can be significantly smaller, with the use of some indirection to use the unsigned shorts as a handle to a string.

The following graphs show the performance of a version using string pointers:

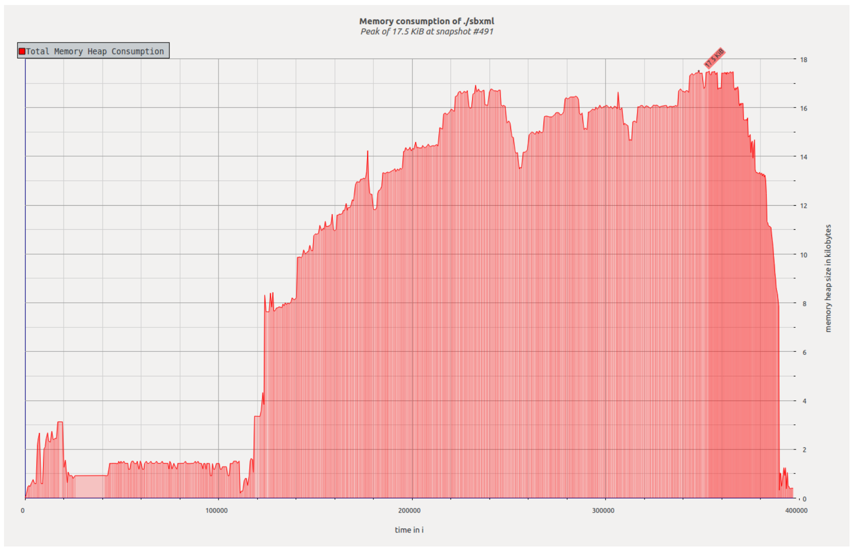

Figure 7: Parsing the small test case using string pointers: 19.55 kb in 387.7 thousand instructions

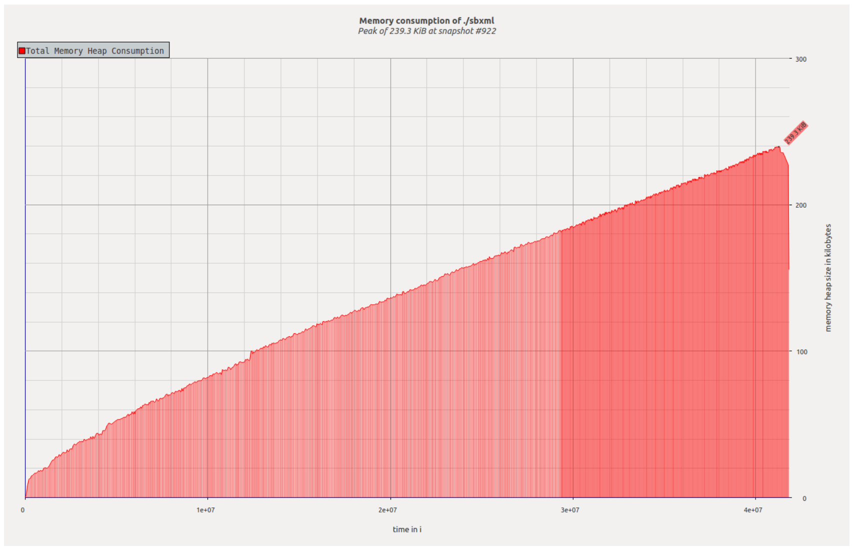

Figure 8: Parsing the medium test case using string pointers: 260.1 kb in 39.89 million instructions

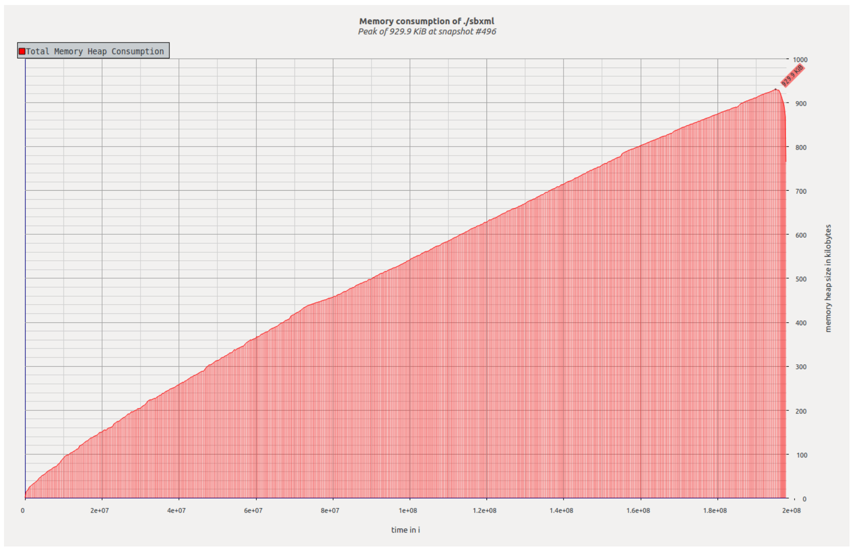

Figure 9: Parsing the big test case using string pointers: 481.0 kb in 188.7 million instructions

The final implementation used 66%, 53%, and 54% less adjusted memory respectively. This points to the final implementation using roughly half the memory of the version with pointers to strings. This can be attributed to the.

Using string handles slows down performance by approximately 1.12, 1.26, and 1.33 times for the test cases small, medium, and big respectively. This is attributable to the layer of indirection required to use handles as opposed to the relatively direct access given by pointers.

While using pointers is faster, it is at best only 33% faster. The memory usage on the other hand almost doubles when using pointers. Even if memory efficiency were not more important than speed, the memory wins here are still larger than the speed impact. Therefore, using string handles is the efficient choice.

2.3.1 Maximum Usable Strings

It should be noted that on the majority of systems, a pointer will be able to handle more strings than an unsigned short. The ANSI C89 standard only guarantees that an unsigned short can hold up to USHRT_MAX, which is $65,535$. This means that at most that many strings can be used. With reference counting, this can be improved to $65,535$ unique strings, which is an important difference as many tag names and attribute names are the same in XML files.

In order to lessen the constraints of this limitation, the system has been designed to use constants for all single character strings and the empty string, which can be common as single character XML tokens such as '<', or with this specific file format, the attribute name "k". This means that the limit can be improved to $65,535$ unique or not unique strings at least 2 characters long.

Analysis of all available samples shows that the maximum number of strings used is in the big test case, with $72,628$ strings at least 2 characters long. This is unfortunately over the maximum limit, although it is an outlier, as the next largest sample uses only just over $24,000$ strings, which is well within the limits.

While going over the limit is unacceptable, reference counting can be used to further improve on the usage. With reference counting, even the big test case is reduced to $3,007$ unique strings at least 2 characters long, and still uses the most. The majority of other cases fall under $1,000$ such strings. This allows for the file format to grow up to 24 times larger, while maintaining the same rate of unique strings, before the limit is reached.

If such a limit is ever reached, changing to a 4 byte data type would still improve memory usage on systems with 8 byte pointers, but considering the focus on embedded systems which are unlikely to be 64-bit, it is unlikely to improve memory usage on most target systems. Therefore it would be best to switch back to using pointers for the speed benefit.

2.4 XML Structure

One of the difficulties of representing the structure of XML, is representing the relationship between nodes, namely, each node has to know which nodes belong to it, in which order. The simple solution is to store a list of pointers to children. A more complex implementation takes advantage of the fact that XML nodes are read in order, meaning that the order the nodes are allocated is the same as the order the nodes should be in. This allows a node to track only the number of descendants it has, direct or otherwise. This takes advantage of allocation order to implicitly define the structure of the XML file.

The pointer based version was inefficiently implemented for speed, and as such the speed should be ignored. The following is the performance for the pointer based version:

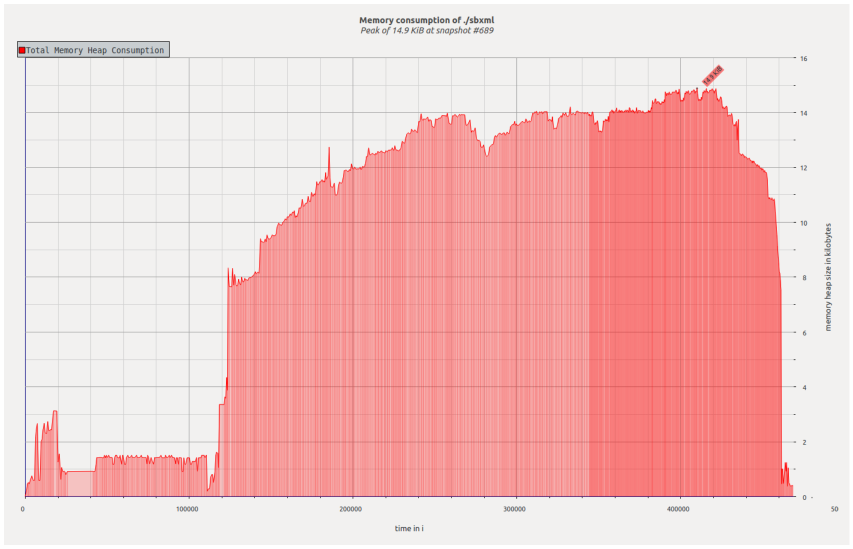

Figure 10: Parsing the small test case using structure pointers: 17.36 kb in 457.7 thousand instructions

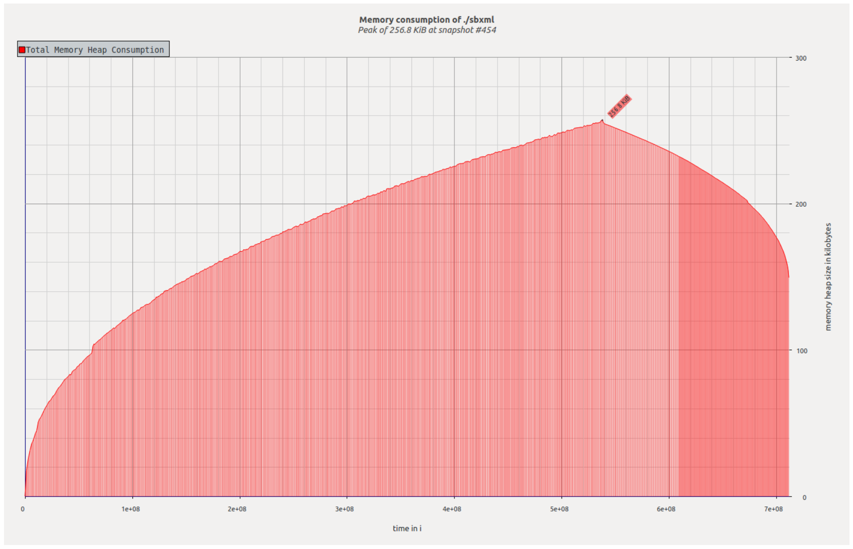

Figure 11: Parsing the medium test case using structure pointers: 355.0 kb in 678.9 million instructions

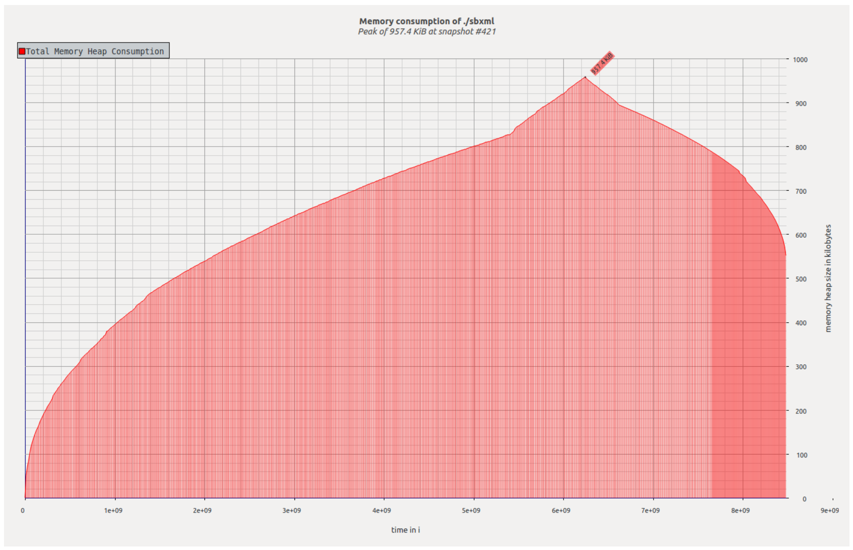

Figure 12: Parsing the big test case using structure pointers: 1,183 kb in 7.897 billion instructions

As can be seen, the pointer based version consistently uses significantly more memory, 1.21, 2.55, and 2.22 times more memory for the small, medium, and big test cases respectively.

The cause of this comes down to the extra memory required to store pointers to children. At best, a node will have no children, and so only requires the storage space for a NULL pointer signifying an empty list. When children are added, this requires a pointer to the list, along with a pointer for each child. Since there is only one node which is not a child of another, and each node must have at least the pointer to the list, this approach will use about 2 pointers for every node. The final implementation on the other hand, uses only one unsigned short for every node. Even on a system with 2 byte pointers, this still means the final implementation uses half the memory, and systems with larger pointers will only save more memory.

While using pointers efficiently would likely improve the speed of the XML parser, the current speed is already acceptable, and having to, at a minimum, double the space required to define the XML node hierarchy is an unacceptable price to pay. Therefore with the interests of memory efficiency in mind, not using pointers is the more efficient choice.

2.5 Attribute Name Sets

After analyzing the data being stored in XML nodes, it occurred to me that there was a major source of duplication amidst the data. The majority of XML nodes could be grouped into a small number of categories which all shared the same attribute names. Yet every node stored the names of their attributes individually. This was changed to instead record which set of attributes the node has, and only store the attribute names once.

The graphs below show the performance of a version which stores attribute names individually in each node:

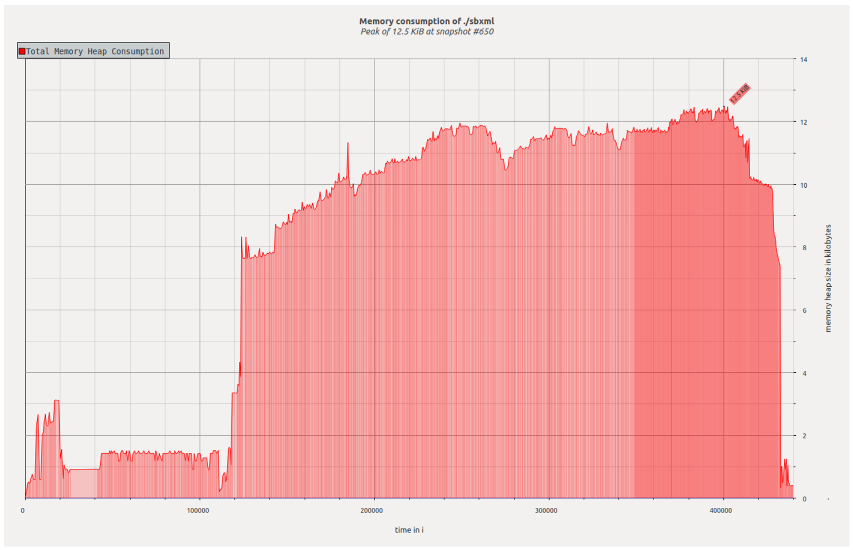

Figure 13: Parsing the small test case while storing attribute names in each node: 14.44 kb in 429.6 thousand instructions

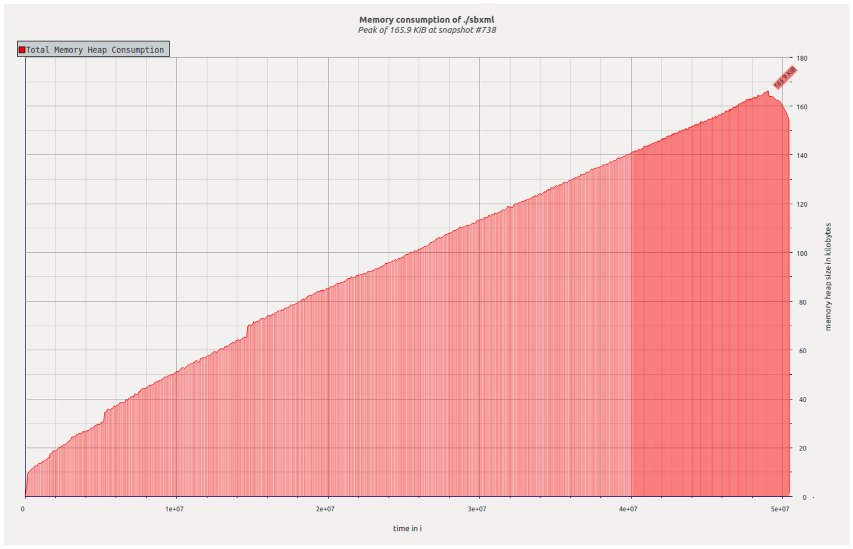

Figure 14: Parsing the medium test case while storing attribute names in each node: 186.6 kb in 48.13 million instructions

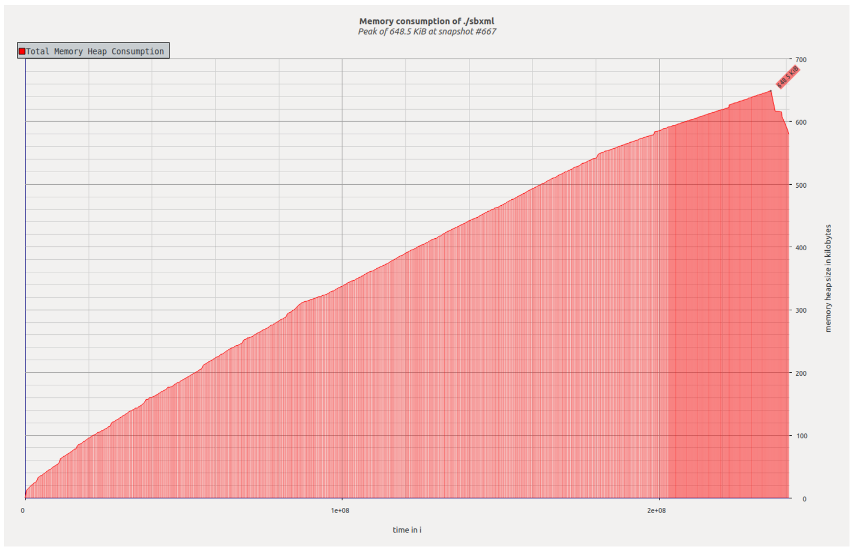

Figure 15: Parsing the big test case while storing attribute names in each node: 722.4 kb in 229.8 million instructions

The small test case uses 99% of the memory with the final implementation, and runs 1% slower for it. The medium and big test cases use 75% and 74% of the memory of the individually stored attribute names version respectively, but run 4% and 9% slower respectively for it. The better memory usage can be put down to no longer storing duplicate attribute names, and the slower runtime to having to access the attribute names indirectly as opposed to directly from the node.

The improved memory usage far outweighs the performance cost, and as such, the efficient choice is to store only unique attribute name sets.

2.6 Batch Allocators

A few types of data structures in the XML representation end up with a significant number of instances, to that end, there is a question of whether it is better to allocate them individually or separately.

The following versions have decreased memory usage in the final implementation that may be unexpectedly large, as technically the same amount of memory should be allocated, if not slightly more in the final implementation for the bookkeeping required for batch allocation. The explanation for this lies with the way that malloc works.

The following two comparisons will focus on wasted memory, which refers to unusable memory created by malloc, either from fragmentation, or the extra memory that malloc allocates for its own reasons.

2.6.1 Attribute Value Allocation

One of the more common allocations required by the XML parser, is allocating an array to hold the attribute values for a node. Since the majority of XML nodes need such an array, they make a good candidate for batch allocation. The final implementation allocates these arrays in batches based on the size of the required array, maintaining separate batch allocators for distinct array sizes.

The graphs below show the performance of a version which individually allocates attribute value arrays:

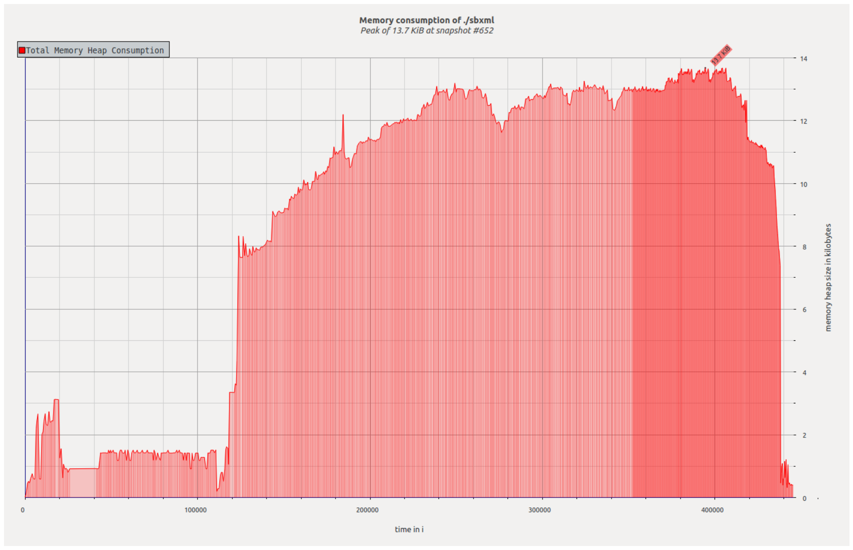

Figure 16: Parsing the small test case while individually allocating attribute value arrays: 16.02 kb in 435.1 thousand instructions

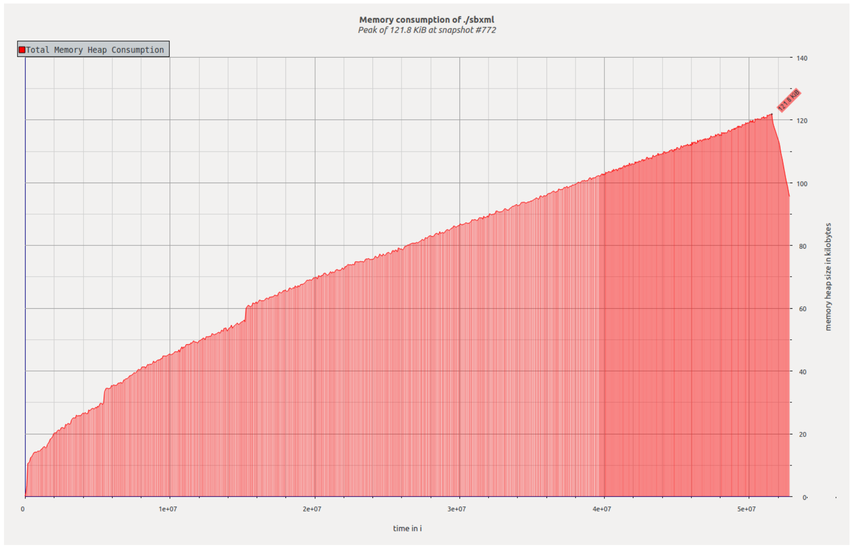

Figure 17: Parsing the medium test case while individually allocating attribute value arrays: 265.6 kb in 50.3 million instructions

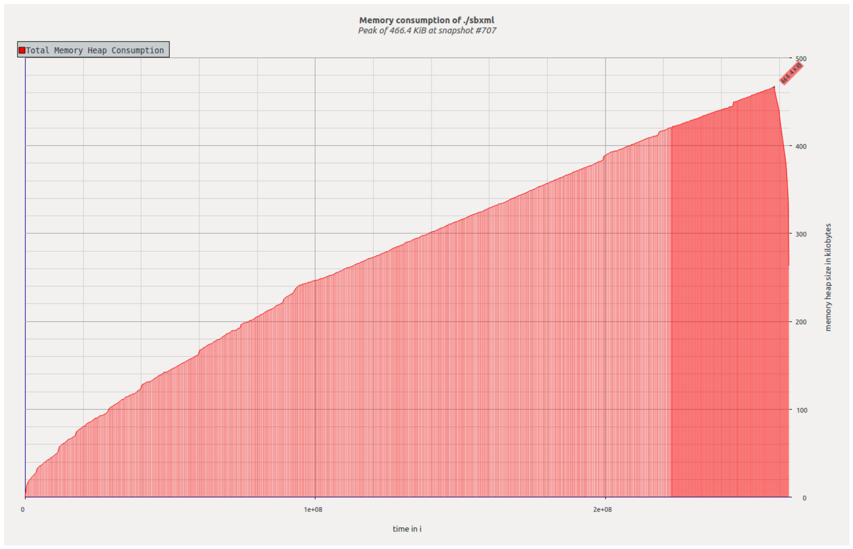

Figure 18: Parsing the big test case while individually allocating attribute value arrays: $1,005$ kb in 251.2 million instructions

The small test case in the final implementation uses 89% of the memory of the individually allocated version, and runs in almost the same number of instructions. The medium and big test cases use 63% and 66% of the memory and also run in almost the same number of instructions.

The fact that both versions complete in almost the same number of instructions indicates that individually allocating each array of values requires approximately the same number of instructions as the system set up to allocate them in batches.

As expected, the difference in memory is significantly larger than would otherwise be expected from a system which should theoretically use more memory. This is all down to wasted space, and on the three test cases the individually allocated version wastes at peak memory usage 2,388 bytes, 102,944 bytes, and 354,485 bytes of memory respectively. While there is very little difference for the small test case, the medium and big test case waste almost five times more memory with individual allocation.

Since the difference in speed is essentially non-existent, and batch allocating the value arrays prevents a significant amount of wasted space, batch allocation for value arrays is the efficient choice.

2.6.2 XML Node Allocation

Another frequently allocated object in the XML representation is the XML node. The final implementation allocates large pools of them at once instead of individually allocating them. Due to implementation constraints, this feature had to be built on top of the version used in the XML Structure section, which uses pointers to track children of nodes. As a result, the memory difference is larger than it otherwise would be, and due to the inefficient speed implementation of that version, the speed of this version must likewise be ignored.

The graphs below show the performance of a version which individually allocates XML nodes:

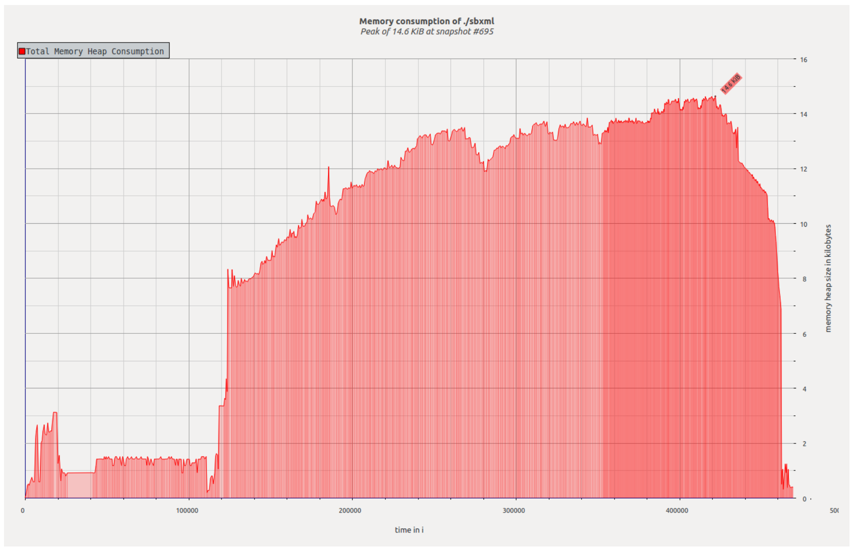

Figure 19: Parsing the small test case while individually allocating XML nodes: 17.41 kb in 458.4 thousand instructions

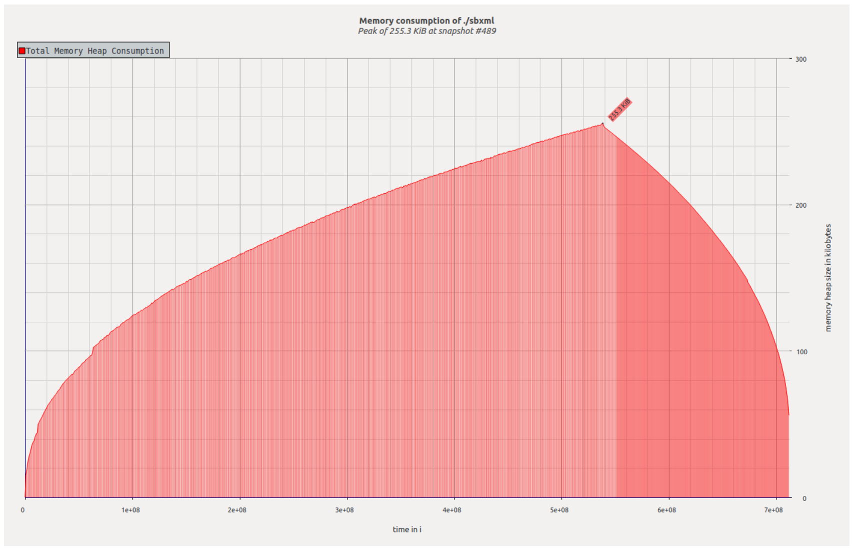

Figure 20: Parsing the medium test case while individually allocating XML nodes: 396.5 kb in 678.8 million instructions

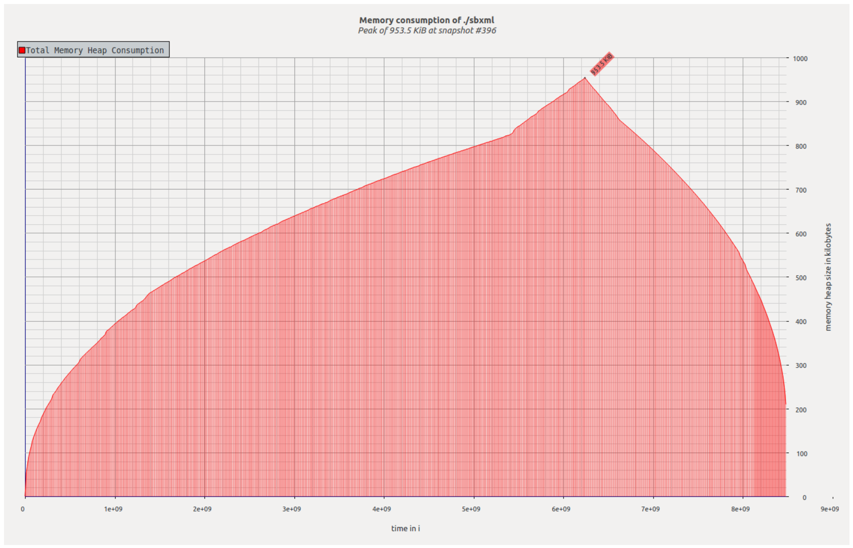

Figure 21: Parsing the big test case while individually allocating XML nodes: 1, 402 kb in 7.897 billion instructions

The final implementation uses significantly less memory in each test, however the memory usage still needs to be adjusted to account for the individually allocated version being built on top of another version. For the adjustment, the difference between this version and the version used for the XML Structure section will be used as the difference between this version and the final implementation. To that end, the adjusted memory usage is as follows:

Small: 14.35 KB, with a difference of 0.050 KB, and an adjusted 2, 479 bytes of wasted space, with a difference of 313 bytes

Medium: 180.6 KB, with a difference of 41.50 KB, and an adjusted 85, 820 bytes of wasted space, with a difference of 64, 520 bytes

Big: 752.8 KB, with a difference of 219.0 KB, and an adjusted 309, 731 bytes of wasted space, with a difference of $234,248$ bytes

This works out to the final implementation using approximately the same amount of memory on the small test case, and approximately 77%, and 71% of the memory for the medium and big test cases respectively. While the difference in wasted space for the individually allocated version is slightly larger than the actual difference in memory usage, this can be attributed to trivial differences in implementation and inaccuracy from the rough adjustment.

It can be inferred from the section on Attribute Value Allocation that the difference in speed between batch allocation and individual allocation for nodes should be insignificant. Based on the rough adjustments, batch allocation will also likely save an estimated 20% in wasted memory. Therefore, batch allocating XML nodes is the efficient choice.

2.7 Original Version

The graphs below show the performance of the original XML parser.

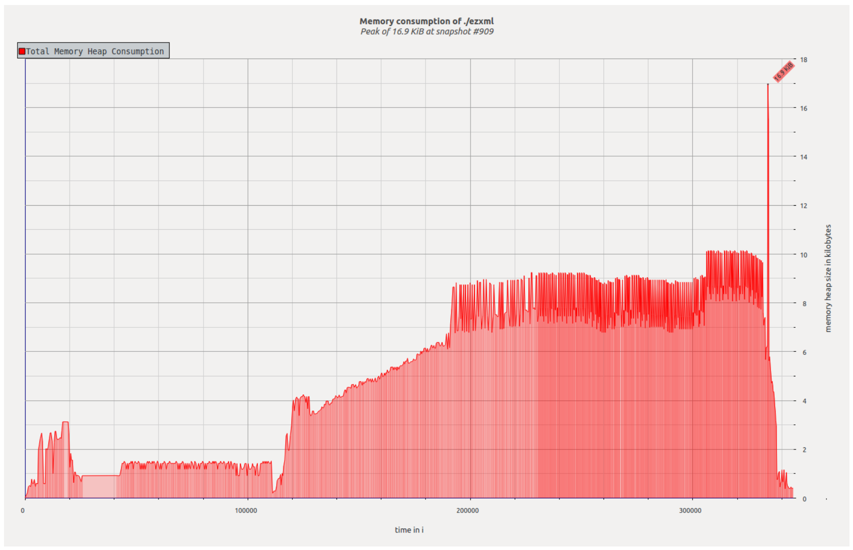

Figure 22: Parsing the small test case with the original XML parser: 17.87 kb in 337.1 thousand instructions

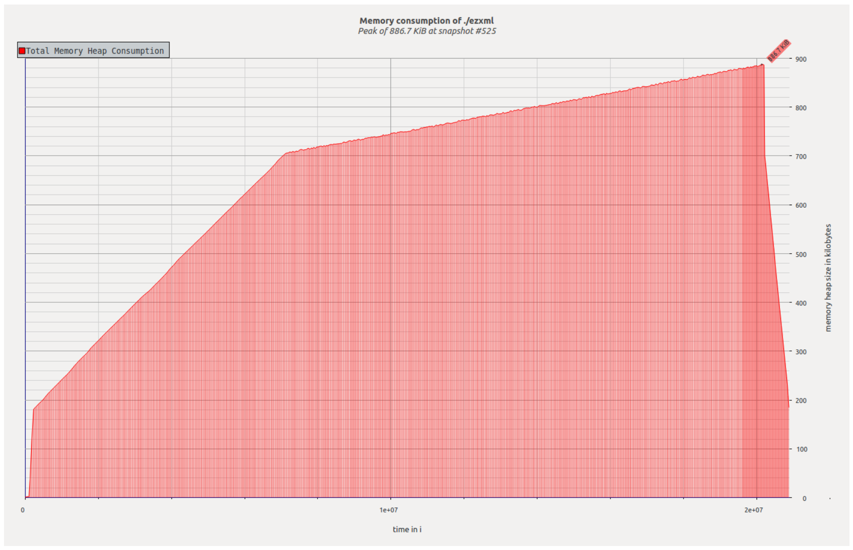

Figure 23: Parsing the medium test case with the original XML parser: 1, 015 kb in 19.92 million instructions

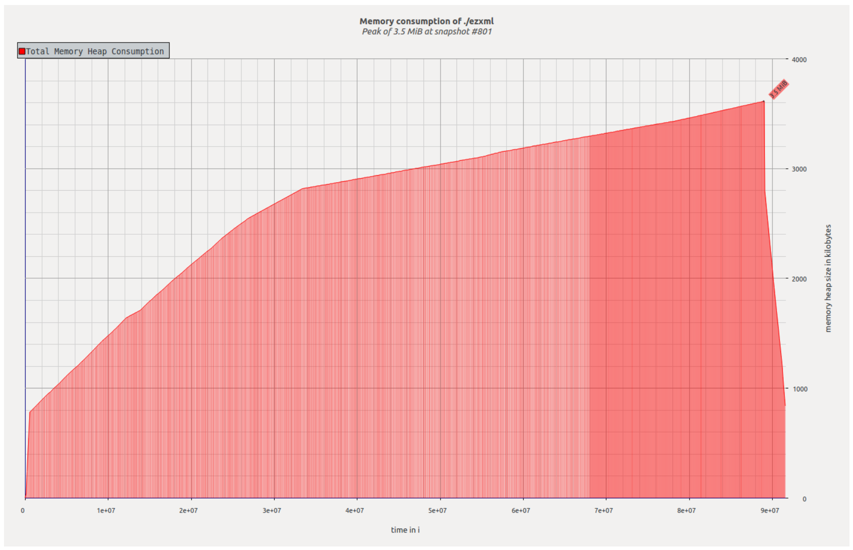

Figure 24: Parsing the big test case with the original XML parser: $4,068$ kb in 87.34 million instructions

The final implementation uses less memory in every case, significantly less in the medium and big test cases. On the other hand, the original version is faster in every case. The final implementation parses the small test case 1.29 times slower, but uses 80% of the memory. The medium test case is parsed 2.52 times slower, and uses 14% of the memory. The big test case is parsed 2.87 times slower, and uses 13% of the memory.

This shows that the final implementation runs anywhere from approximately as fast to 3 times slower. This can be attributed to every design choice conceding worse speed for better memory usage. The memory usage of the final implementation uses anywhere from almost the same amount of memory to almost 10 times less memory. Once again attributable to all the design choices being made to minimize memory usage.

While the final implementation is slower than the original XML parser, the speed difference is less than 3 times slower, while the memory difference is 10 times less memory used by the final implementation. Since the original XML parser was already extremely fast, 3 times slower is an acceptable speed. The significantly reduced memory usage is in line with the goal to use less memory. Overall, the final implementation is the more efficient choice, especially as limiting memory usage is more important than speed.

3.0 Conclusions

Based on the above analysis, the most significant improvement in terms of memory usage, was reference counting strings, as this removed a massive amount of data duplication. This allowed the XML parser to use almost a fifth of the memory that would otherwise be required.

The next best improvement came from using an implicit definition of the structure of XML as opposed to pointers to children, which removed the need for a massive number of pointers. The result is that the final implementation uses less than half of what would otherwise be required.

Using string handles instead of pointers is a seemingly small change which still managed to cut memory usage almost in half thanks to the much smaller size of the handle compare to a pointer.

Due to having to allocate an array of attribute values for almost every node in the XML file, allocating these arrays in batches reduced memory to about 75% of what it would otherwise be. This is primarily thanks to reducing the wasted space that quickly grows with individual allocations .

Similarly, batch allocating the XML nodes themselves netted a similar memory reduction to about 74% of what would have been used. Once again showing that large amounts of individual allocations start to waste memory very quickly.

Finally, storing only unique attribute name sets reduced duplication in the

XML nodes, allowing for a 75% reduction in memory used. This is attributable to storing attribute name sets in a central location instead of each node storing the same set of names again and again.

Based on the goals of memory efficiency and speed, the various versions of the XML parser were judged based on their memory used and instructions required to parse input for three representative test cases. The memory usage was the most important concern, with a focus on limiting the number of allocations. The speed of the XML parser, measured in instructions for convenience, is a secondary concern, as the XML parser still has to be fast.

The final implementation of the efficient XML parser runs at worst, three times slower than the original XML parser, but makes up for it by using up to ten times less memory. The final implementation surprisingly manages to reliably use less memory than the size of the input file to parse and represent the XML within.

4.0 Recommendations

While the final implementation of the efficient XML parser is an improvement on the original version, there is still room for improvement.

The final implementation can run up to three times slower than the original, and while the original was extremely fast, it would still be great to improve the speed of the new parser. Reference counting strings, while a great win for memory usage, had a significant impact on speed. Not only does it require a significant amount of comparisons for each new string, the number of comparisons required only increases as the input grows larger. While this was implemented using basic hashing to limit the number of comparisons required, this could still be improved, possibly using a more advanced system such as a Trie or Patricia Tree.

If more speed is required, the final implementation was not optimized using code profiling, and the use of such tools may point to sections of code which could be rewritten for improved performance.

If further improvements to memory efficiency are required, it may be possible that the unique attribute name sets may also match up with unique tag names, such that in most cases, two tags with the same name share the same attribute names. This would allow the tag name to be stored with the attribute name sets to remove a string handle from the node data structure.

References

1 Crank Software Inc.: About. Retrieved 16 December, 2015 from `http://www.cranksoftware.com/about`

2 Extensible Markup Language: Introduction. Retrieved 16 December, 2015 from `http://www.w3.org/XML/`

3 ANSI C89. Retrieved 16 December, 2015 from `http://port70.net/~nsz/c/c89/c89-draft.html`

4 Valgrind Memcheck: User Manual. Retrieved 16 December, 2015 from `http://valgrind.org/docs/manual/mc-manual.html`

5 Valgrind Massif: User Manual. Retrieved 16 December, 2015 from `http://valgrind.org/docs/manual/ms-manual.html`

6 Massif Visualizer: Overview. Retrieved 16 December, 2015 from `https://projects.kde.org/projects/extragear/sdk/massif-visualizer`